AN ABSTRACT OF THE THESIS OF

Santhosh Kumaran for the degree of Doctor of Philosophy in Computer Science presented on July 26, 1996.

Title: Towards Architecture-Adaptable Parallel Programming.

Abstract approved: Redacted for Privacy

Michael J. Quinn

There is a software gap in parallel processing. The short lifespan and small installation base of parallel architectures have made it economically infeasible to develop platform-specific parallel programming environments that deliver performance and programmability. One obvious solution is to build architecture-independent programming environments. But the architecture independence usually comes at the expense of performance, since the most efficient parallel algorithm for solving a problem often depends on the target platform. Thus, unless a parallel programming system has the ability to adapt the algorithm to the architecture, it will not be effectively machine-independent.

This research develops a new methodology for architecture-adaptable parallel programming. The methodology is built on three key ideas: (1) the use of a database of parameterized algorithmic templates to represent *computable* functions; (2) frame-based representation of processing environments; and (3) the use of an analytical performance prediction tool for automatic algorithm design.

This methodology pursues a problem-oriented approach to parallel processing as opposed to the traditional algorithm-oriented approach. This enables the

development of software environments with a high level of abstraction. The users state the problem to be solved using a high-level notation; they are freed from the esoteric tasks of parallel algorithm design and implementation.

This methodology has been validated in the format of a prototype of a system capable of automatically generating an efficient parallel program when presented with a well-defined problem and the description of a target platform. The use of object technology has made the system easily extensible. The templates are designed using a parallel adaptation of the well-known divide-and-conquer paradigm.

The prototype system has been used to solve several numerical problems efficiently on a wide spectrum of architectures. The target platforms include multicomputers (Thinking Machines CM-5 and Meiko CS-2), networks of workstations (IBM RS/6000s connected by FDDI), multiprocessors (Sequent Symmetry, SGI Power Challenge, and Sun SPARCServer), and a hierarchical system consisting of a cluster of multiprocessors on Myrinet.

Towards Architecture-Adaptable Parallel Programming

by

Santhosh Kumaran

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed July 26, 1996
Commencement June 1997

Doctor of Philosophy thesis of Santhosh Kumaran presented on July 26, 1996

APPROVED:

Redacted for Privacy

Major Professor, representing Computer Science

Redacted for Privacy

Chair of the Department of Computer Science

Redacted for Privacy

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Santhosh Kumaran, Author

ACKNOWLEDGMENT

I am grateful to my advisor, Professor Michael Quinn, and the rest of the committee for their guidance. This research would never have materialized had Professor Quinn not accepted me as his student. He provided me with financial assistance in the final phases of this project, enabling timely completion of this research. Much of what I am today is due to his advice and encouragement in the last four years.

I was very fortunate to have Professor Tom Dietterich in my committee. His insightful comments have contributed a great deal towards the success of this project.

I would have probably never done a Ph.D. had Professor Miller not hired me as an RA in Oceanography. The seeds of this research were sown while working with Professor Miller, applying parallel processing for oceanographic problem solving.

I am grateful to Chandra Reddy and Jason Moore for the discussions and debates, which helped in fleshing out many of the ideas in this thesis.

This research would not be possible without the love and understanding of my wife, Joyce Federiuk.

I am grateful to my parents for teaching me to dream and to work hard to make those dreams come true.

This work was supported by NSF grant ASC-9208971.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF TABLES

DEDICATION

To Joyce, of course.

TOWARDS ARCHITECTURE-ADAPTABLE PARALLEL PROGRAMMING

1. INTRODUCTION

This research is inspired by my experience in using parallel processing for several scientific applications at the College of Oceanic and Atmospheric Sciences (COAS) at Oregon State University. During the relatively short period of four years I was there, the field of parallel processing fell from the upper echelons of success and respect to the trenches of failure and neglect. The two quotes below reflect the mood change better than a thousand words:

"Parallel processing works!" –Geoffrey Fox (1991)

"Did the MPP bandwagon lead to a cul-de-sac?" –Gordon Bell (1996)

What went wrong? I will try to answer this question based on my own experience using parallel processing for real-world problem solving. In the process, I will also lay out the philosophical foundations of this dissertation. I will begin with a few observations:

- The change in the parallel processing infrastructure at COAS over a short time span is testimony to the state of flux of parallel architecture. The parade of machines that came through the doors of the COAS computing center in four years includes:

- CM-200
- CM-5
- Network of Workstations (IBM RS/6000s on FDDI)
- SGI
- IBM SP2

What do these platforms have in common? Not much, except that they are all built by companies with three-letter names.

- It was necessary to rewrite application programs to run efficiently on the new platforms. Complex numerical models typically require several years to develop [1]. Thus, it is obvious that developing non-trivial applications for a specific parallel platform is not very practical.

- High level languages such as Fortran 90 provide portability, but typically they have been able to harness less than 5% of the computing power of the parallel machines [1]. Thus, it was not uncommon for the sequential program run on a "killer workstation" to beat the parallel implementation of that program on the parallel computer.

- Developing parallel programs for complex applications remains an esoteric task, even with the availability of high level languages. For example, COAS has some of the best scientists in the world and several applications which are begging for parallelization. But the fraction of people who actually use these machines is disappointingly low.

- There is a staggering amount of research being done on parallel programming paradigms and languages. But the impact of this on practical parallel processing has been minimal. In fact, the biggest impact resulted from a rather simple

idea: using the data-parallel paradigm for developing parallel algorithms and languages across all parallel architectures [2].

- Though the scientific community has steadfastly stuck to Fortran as the programming language of choice for decades, something very interesting happened in the '90s. These users enthusiastically embraced MATLAB, a problem solving environment (PSE) for numeric computing [4].

The common refrain among computer scientists regarding the fiasco of parallel processing goes like this: there are not enough applications; parallel computing is a solution waiting for a problem [3]. I believe this could not be farther from the truth. Given the poor performance, lack of portability, and the difficulty of programming, most users wisely stayed away from parallel processing. The problem is not the lack of applications, but the lack of a viable solution.

1.1. Philosophical Foundations

The philosophical foundations of this research are derived directly from the observations above:

- Most users do not wish to do any programming, let alone parallel programming. (Thus, what we need is not High Performance Fortran, but High Performance MATLAB.)

- Architecture adaptability is an extremely important property in parallel programming systems.

- If necessary, generality should be traded off for architecture adaptability, performance, and ease of use. Most users employ their computers for problem

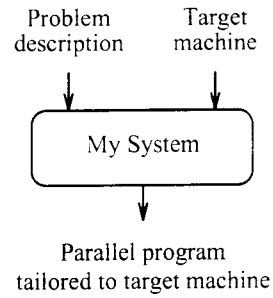

FIGURE 1.1. Box view of the system.

solving in a specific application domain. It is scientific computing at COAS, it might be data mining at a Wall Street firm, and it might be database applications at a bank.

1.2. Problem Statement

Thus, I set out to build a problem solving environment (PSE) for parallel and distributed computing. I wanted my system to be architecture-adaptable, but restricted the problem set to be domain-specific. Scientific computing is the application domain. Figure 1.1 shows the system functionality schematically.

The system has the following inputs:

- **Problem description**: The user describes the problem to be solved using a high-level notation. The notation to be used will depend on the target application domain. For scientific computing, the notation used in MATLAB is a good candidate.

- **Target machine**: We require the user to merely identify a particular machine. The system keeps a database of processing environments, and uses the identifier as an index to retrieve the required information from the database.

  It is straight forward to extend the system capability in such a way that it would not be necessary for the user to specify a target machine at all. By storing the information on machines available to each user, the system could search for the best machine to solve the given problem.

The output from the system is a parallel program that solves the problem efficiently on the target machine.

There are no such systems in existence today. More importantly, no published work explains how such a system may be realized. This dissertation aims to develop a methodology which can be used to build systems of this nature for scientific computing.

1.3. Research Contributions

Most of the research contributions in this dissertation are already in print as refereed conference papers or journal articles, and the rest are being reviewed. The best way to present them is by listing the papers and summarizing their contents:

- *Divide-and-Conquer Programming on MIMD Computers*, **International Parallel Processing Symposium (IPPS'95)** [5]: In this paper, we present a template for parallel processing based on the divide-and-conquer paradigm (PDC-template). We show how good performance is obtained for several linear algebra kernels on shared-memory as well as distributed-memory MIMD machines using the template approach. The key to efficiency is our introduction

of a new and innovative scheme to implement divide-and-conquer algorithms in the parallel environment.

- *On Parallel Divide-and-Conquer*, **Transactions on Parallel and Distributed Systems** [6]: *Divide-and-conquer* is a very old concept, much older than computer science. Julius Caesar used divide and conquer to subdue Gaul in first century B.C. (Some think that is where the phrase came from, probably due to his famous line: "I came, I saw, I conquered.") But our approach to divide-and-conquer is very different from any previous work. Consequently, we are able to get some new and interesting theoretical results about the power of divide-and-conquer. In this paper, we present the theoretical foundations of our work. In particular, we discuss the following issues:

  - A large body of legacy code exists in Fortran for scientific applications. Additionally, application developers in this domain are very familiar with Fortran. So, a parallelizing compiler for Fortran sounds like an attractive alternative approach to solving the problem this dissertation addresses. I use computational complexity theory to argue against an approach based on parallelizing compilers. I show that the task of a general purpose parallelizing compiler is undecidable even if we have an oracle that solves the halting problem.

  - How hard is optimal divide-and-conquer? I prove that optimal divide-and-conquer is NP-hard. This result is important in formulating the methodology, since it implies we need to focus on good heuristics to partition the problem.

  - What problems can be efficiently solved using my methodology? The case studies I present in this work give strong evidence of the power of the

system, but this evidence is empirical. I use the circuit model to prove that any parallel computation can be formulated using our computational model without changing its complexity class. In particular, if the original problem is in NC^k, then the formulation using our model is also in NC^k.

- *Architecture-independent Parallel Programming using the Divide-and-Conquer Paradigm*, **Third Workshop on Languages, Compilers, and Run-time Systems (LCR)** [7]: We propose a methodology for architecture-adaptable parallel processing based on the PDC-template. We suggest that architecture adaptability may be realized by parameterizing the template and selecting appropriate values for these parameters based on analytical performance prediction.

- *Towards Architecture-Adaptable Parallel Programming*, **Scientific Programming** [8]: This journal article builds on the previous conference papers. We introduce the notion of a composite template and show how they can be used to represent complex applications. We map the conjugate gradient method onto three diverse processing environments using our methodology. The results demonstrate the applicability of our methodology for architecture-adaptable parallel processing.

- *Architecture-adaptable Finite Element Modeling: A Case Study using an Ocean Circulation Simulation*, **Supercomputing 1995 (SC'95)** [9]: In this paper, we use our methodology for architecture-adaptable finite element modeling. Finite element methods play a significant role in the modeling of physical systems. We show how a template-based approach can be used to automate the mapping of a finite element model to a multicomputer as well as to a workstation network.

- *Architecture-Adaptable Parallel Processing using Object Technology*, **Parallel Object-Oriented Methods and Applications Conference (POOMA'96)** [10]: This paper describes the design and implementation of a research prototype of a system capable of automatically generating efficient parallel programs for diverse architectures. We use object technology for developing the system.

- *Automatic Exploitation of Dual Level Parallelism on a Network of Multiprocessors*, **International Symposium on High Performance Distributed Computing (HPDC-5)** [11]: Most of the parallel processing environments in use when this research was proposed could be categorized either as a multicomputer, a multiprocessor, or a workstation cluster. Hence, the target architectures we had considered up to this point were from these categories. But a new architecture has emerged while the project has been coming to a close: a network of shared-memory multiprocessors. Obviously, the ability of my methodology to handle the new architecture is a good test of its validity, especially since the main purpose of my work is to provide architecture adaptability.

  This paper shows how my methodology was gracefully extended to solve problems efficiently on an SMP network. We use the Kalman Filter [12], a widely used data assimilation scheme, as the test application. The processing environment consisted of four SUN SPARCServer 20s on Myrinet [13]. Illinois Fast Messages (FM) [49] was used for message passing.

- *An Architecture-Adaptable Problem Solving Environment for Parallel and Distributed Computation*, **Journal of Parallel and Distributed Computing** [15]: This journal article is built on the SC'95, POOMA'96, and HPDC-5

conference papers. We describe the system in detail and show how it can be used to map complex scientific applications to diverse architectures. Target platforms include a multicomputer, a multiprocessor, workstation network, and a cluster of SMPs. We also describe how a high level user interface—web-based or MATLAB style—can be integrated into the system to provide users with the benefits of *metacomputing* without its hardships.

1.4. Dissertation Organization

This thesis is presented in manuscript format, consisting of three journal articles. In this chapter, I have summarized the contents of these journal articles. The second chapter presents the theoretical foundations. It consists of the paper "On Parallel Divide-and-Conquer," to be submitted to the *Transactions on Parallel and Distributed Systems.* The third chapter gives the methodology and proof of concept. It consists of the journal article "Towards Architecture-Adaptable Parallel Programming," accepted for publication in the journal of *Scientific Programming.* The fourth chapter discusses the system integration and validation. It consists of the paper "An Architecture-Adaptable Problem Solving Environment for Parallel and Distributed Computation," to be submitted to the *Journal of Parallel and Distributed Computing.* A discussion of the results, future work, and the project summary appears as chapter 5, the concluding chapter.

2. THEORETICAL FOUNDATIONS

On Parallel Divide-and-Conquer

Santhosh Kumaran and Michael J. Quinn

Submitted to *IEEE Transactions on Parallel and Distributed Systems*
IEEE Computer Society
New York, NY

2.1. Abstract

We present the theoretical foundations of a methodology for architecture-adaptable parallel processing. We use parallel divide-and-conquer (PDC) as our computational model in developing the methodology. By employing an innovative approach to implement parallel programs within the divide-and-conquer paradigm, we argue that PDC is a powerful tool for developing architecture-adaptable, efficient, and easy-to-use parallel programming systems. We show optimal PDC is NP-hard, which leads us to domain-specific heuristics in developing the methodology. We apply the circuit model to prove that any parallel computation can be formulated within our computational model without changing its complexity class. In particular, if the original problem is in NC^k, then the formulation using our model is also in NC^k. This result indicates we are not losing much in expressibility due to the restricted form of our computational model. We demonstrate the applications of our computational model by implementing a set of linear algebra kernels on a multiprocessor and a multicomputer.

2.2. Introduction

The crisis in parallel processing software can be tackled only by having architecture-adaptable parallel processing systems. We present a methodology for developing such a system for the application domain of scientific computing in [8]. Our objective in developing this methodology is to provide to the users a system with the following capability: given a problem instance and the description of a target machine, generate a program that can solve the problem efficiently on the specified machine. In this paper, we explore the theoretical foundations of this methodology. In particular, we discuss the following issues:

- A large body of legacy code exists in Fortran for scientific applications. Additionally, application developers in this domain are very familiar with Fortran. So, a parallelizing compiler for Fortran seems to be an attractive option to solve the problem our system aims at. We use computational complexity theory to argue against an approach based on parallelizing compilers. We show that the task of a general purpose parallelizing compiler is undecidable even if we have an oracle that solves the halting problem.

- Our methodology is built on a computational model based on divide-and-conquer. Since the goal of any parallel processing system is to minimize the execution time, it is desirable for our system to be able to formulate the solutions to the given problem using an *optimal* divide-and-conquer algorithm. Before we attempt to do that, we need to ask the question: How hard is optimal divide-and-conquer? We prove that optimal divide-and-conquer is NP-hard. This result is important in formulating the methodology, since it tells us we need to focus on good heuristics in forming the solution.

- What problems can be efficiently solved using our methodology? The *scope* of the system is defined as the set of problems it can solve efficiently. If the scope is too restricted, then the system obviously has limited use. We use the circuit model to prove that any parallel computation can be formulated using our computational model without changing its complexity class. In particular, if the original problem is in NC^k, then the formulation using our model is also in NC^k.

2.3. Architecture Adaptability in Parallel Processing

First, we need to explore the notion of architecture adaptability further. In general, there are infinitely many algorithms that solve any given problem. We are interested in the algorithm that solves the problem on the given machine in the least amount of time. This algorithm is obviously a function of the target machine. In parallel processing, machine models abound. Hence, the algorithm design is an important part of problem solving in the parallel environment.

Solving a problem on a computer is the same as computing a function. An architecture-adaptable system is one which selects the *best* algorithm automatically to compute the given function on the specified platform. There is an obvious problem here: for several problems of practical importance, we do not know what the best algorithm is. Thus it is impossible to build a system which is fully architecture-adaptable. But a restricted form of architecture adaptability can be achieved if the system is able to select the best one from among a set of algorithms to compute the function.

2.3.1. Architecture Adaptability and Parallelizing Compilers

The traditional approach to problem solving on computers employs a programming language to "code" an algorithm that computes the function we are interested in. The compiler translates the algorithm from the source language to the machine language of the target platform. In sequential computing, this works fine since the RAM is a good model of single-CPU computers. A compiler that translates a source program written in a sequential programming language into a form that can be executed on a parallel computer is called a parallelizing compiler. The quality of the parallelizing compiler is measured by the efficiency of the generated code. If we could develop good parallelizing compilers for each one of the available target machines easily, then architecture adaptability would be a moot point. Users would need to write only a single program—a sequential program that solves the problem—and the parallelizing compiler would do the rest of the work.

In the past two decades, a lot of effort has been spent on developing parallelizing compilers; it remains an active research area. The progress made so far in achieving good performance on parallel architectures using this approach is far from satisfactory. Given the amount of research effort expended in this field, the reason for the lack of results can only be explained by the difficulty of the problem. Let us begin by looking at the task of a parallelizing compiler: Given a sequential program, coded in a language such as Fortran 77, the compiler should output an efficient program for the target machine, which is a new parallel architecture. The example given below illustrates the difficulty of this task.

Consider the numerical solution of partial differential equations (PDE), an application area where a tremendous amount of Fortran 77 software already exists. The algorithms embedded in these Fortran programs are targeted for sequential

machines. A good example is the preconditioned global conjugate gradient method (PCG) for solving elliptic equations. Now consider solving the same problem on a parallel architecture. Most state-of-the-art parallel implementations use the multigrid scheme or a domain decomposition method, since they perform much better than PCG methods [1]. So the task at hand may be summarized as follows for this example:

Scan in a Fortran 77 program.
Analyze the code and unearth the fact that
    it is implementing a PCG algorithm.
Understand that multigrid scheme is much more efficient
    for solving the underlying PDE problem
    on the specified target architecture.
Output an efficient implementation of the multigrid scheme
    on the target architecture.

Remember that a similar procedure may have to be carried out for *any* Fortran program. It follows that the *supercompiler* should be able to map from a Fortran program to the *function* the program attempts to implement. But the set of computable functions, though countable, is not *effectively* (computably) countable [16]. In other words, given integers i and j, we cannot compute $f_i(j)$, where f_i is the i^{th} total recursive function. We can show this by using a diagonalization argument. Assume that it is possible to compute $f_i(j)$, given i and j. We can then form the function $f^*(i) = f_i(i) + 1$. This immediately leads to a contradiction: on one hand f^* is not a total computable function since it differs from each f_i in the list, while on the other hand we just showed how to compute it.

The task of the parallelizing compiler as described above and the listing of the total recursive functions are related. Given a Fortran source code as the input, the task of such a compiler may be broken into the following steps:

1. Use the source code to generate an index i.

2. Use this index to determine the i^{th} function f_i, which is the function the source program computes.

3. Generate code to compute this function efficiently on the target platform.

But implementing the second and third steps above in effect will permit us to compute $f_i(j)$, given i and j. (Note that j represents the input to the program and is obtained by mapping N^* to N.) Hence no such mapping can exist and the outline of the supercompiler sketched above is not just difficult, but impossible.

We considered only the total recursive functions in the discussion above. In other words, we limited the input to the compiler to consist only of Fortran programs that always halt. What if we remove this restriction? In general, the problem of generating optimal code for even the *sequential* machines is undecidable when no restrictions are placed on the type of the target processor or on the input program. Assume that we have an oracle that solves the optimal code generation problem of the sequential machine. We can use this oracle to solve the halting problem as well, since the halting problem is at the lowest level in the hierarchy of undecidable problems. Now consider the task of the parallelizing compiler as outlined above and specifically the second step of the task. From the diagonalization argument, it is evident that the parallelizing compiler problem is undecidable even if we are given an *oracle* to solve the optimal code generation problem of the sequential machine.

Notice that the undecidability of the optimization problem does not imply that efficient solutions can not be generated. Optimizing compilers for sequential machines are extremely successful in generating efficient code in most instances. This is accomplished by transforming the code based on some heuristics. (A good example is the *template matching* based heuristics that traditional compilers use for code generation.) These transformations are called "optimizations", which is

a misnomer since, as Aho et. al. [17] pointed out, "there is rarely a guarantee that the resulting code is the best possible". Vectorizing compilers use a set of analogous transformations, most important of which is the vectorization of loops. Loop interchanging, loop skewing, loop fusion and loop concurrentization are also included in this set [18]. In general, these transformations are localized. For the parallelizing compiler to be effective, we need global transformations. We believe that the argument presented above, based on the hierarchy of undecidable problems, highlights the difficulty of finding such global transformations.

Let us consider the task of the parallelizing compiler sketched above once again. Since the user knows the functionality of his sequential program, he can tell the system as much. Thus the task of the system reduces to providing an efficient algorithm and coding it up in a suitable target language, when given the problem instance and the specification of the target platform. Our approach to architecture adaptability is to build such a system.

2.3.2. Our Approach to Architecture Adaptability

We begin by restricting the set of functions we aim to compute. We call this set *primitive* functions. Each primitive function is identified using a name. A typical computation will be a composition of the primitive functions. We call this a *composite* function; it can contain any number of primitive functions.

Suppose we could associate a finite number of algorithms with each primitive function, where each algorithm computes the associated function. We call this set of algorithms the solution space of the function. The solution space of a composite template is the cartesian product of the solution spaces of its constituent templates.

Our approach to architecture adaptability is to search this solution space and select the best algorithm for the specified target machine.

There are two very important issues that we need to address for this approach to be viable.

1. How do we represent the solution space?

2. How do we search the solution space for the best solution?

Our thesis is that the divide-and-conquer problem solving strategy can be used to represent the solution space and analytical performance prediction can be used to select the best template.

2.4. Parallel Divide-and-Conquer

Divide-and-conquer is a well-known problem-solving strategy in which a problem is solved by dividing it into a number of smaller subproblems and then solving the subproblems by the recursive application of the same procedure. Infinite recursion is prevented by using a *base predicate* which triggers a *base function.* The solutions to the subproblems form partial results, which are combined to form the final result. In parallel divide-and-conquer, the subproblems are solved in parallel, providing an easy opportunity for exploiting parallelism in architecture.

A program is represented in our methodology as a sequence of divide-and-conquer operations. Figure 2.1a shows the graphical representation of such a program in the form of a directed acyclic graph (DAG), comprising three divide-and-conquer operations. The shaded squares denote the base cases. Note that the number of subprograms generated and the depth of recursion may change for each invocation of the operation. Essentially, each operation has a well-defined top-level

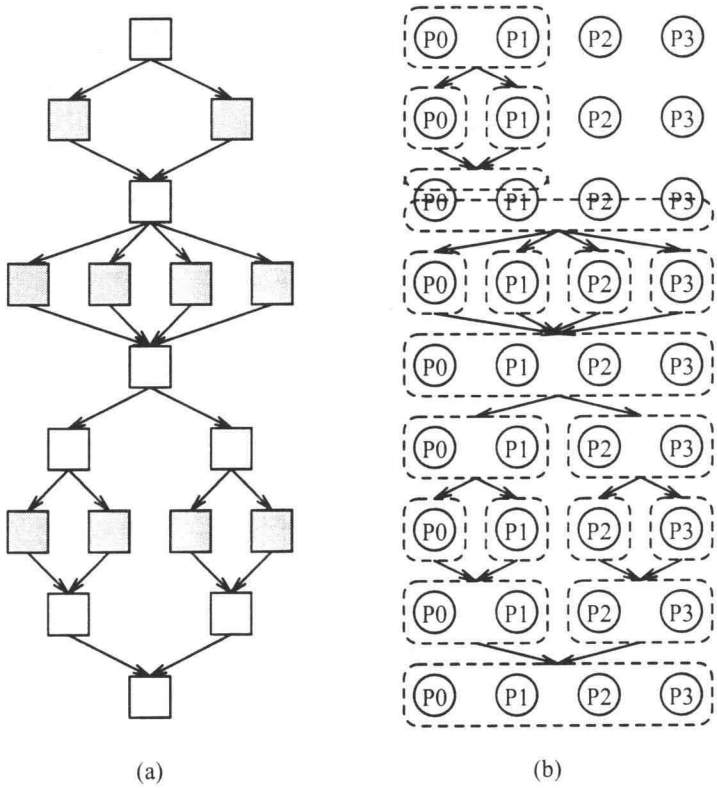

FIGURE 2.1. Mapping of the DC-tree to the processing nodes. Task graph shown on the left is mapped as shown on the right. The shaded squares denote the base cases. Note that the first DC-operation is performed using only two processors while the rest of the computation uses four processors.

structure, but the details can change for each invocation. We use the notion of a *parameterized template* to represent these operations; the template describes the top-level structure and the parameters are used to add the details. Each primitive function and the associated solution space is represented by one or more such templates. Composite-templates, consisting of cascaded divide-and-conquer operations such as the one shown in Figure 2.1a, are formed from these *base* templates. Composite-templates represent the composite functions described in the earlier section.

Our approach to parallel divide-and-conquer is different from others in the mapping of the subproblems to the processing nodes. We combine the divide-and-conquer paradigm with the Single Program Multiple Data (SPMD) style of programming to obtain an efficient implementation of the cascaded divide-and-conquer explained above. The subproblems at each level of the DAG are mapped to all or a subset of the processors. This is in contrast to the conventional approach to divide-and-conquer programming where each subproblem gets mapped to a single processor. Figure 2.1b shows a possible implementation of the program in Figure 2.1a using two processors for the first divide-and-conquer operation and four processors for the rest of the computation.

2.4.1. Divide-and-Conquer Template

Consider an instance of a problem, $O = F(I)$, where the problem is to map from an input data set I to an output data set O. For matrix multiplication ($C = AB$), set $I = \{A, B\}$, and set $O = \{C\}$.

The divide-and-conquer template can be expressed in a functional form. Let us call this function $DC()$. Thus solving a specified problem, $O = F(I)$, on the target environment reduces to invoking $DC()$ with proper arguments. The arguments to $DC()$ are the following:

1. A *divide* function $D()$, which decomposes the input data set into a set of subdomains.

   $D() : I \rightarrow \{I_1, I_2, \ldots, I_n\}$

2. A *conquer* function $C()$ which is applied on each of the subdomains.

   $C() : I_i \rightarrow O_i; i \in \{1, \ldots, n\}$.

3. A *combine* function D^{-1}, which is the inverse of the divide function.

   $D^{-1} : \{O_1, O_2, \ldots, O_n\} \rightarrow O$.

Note that the conquer function may be the same as F, which leads to a recursive definition. To prevent infinite recursion, a *base predicate* and a *base function* are used as mentioned earlier. A constraint may also be specified on the order in which the subproblems are solved, but in most applications, no constraints are present and the tasks can be executed in parallel.

2.4.2. Complexity of the Problem

The divide function is key to the whole problem. The combine function and the conquer functions are dependent on it. The performance on any processing environment is directly related to the divide function. Ideally, we would like to have an optimal and generic divide function. Such a function would decompose the problem so as to minimize the execution time of the given problem instance on the specified target environment. Suppose we have access to such a function; we call it D_{opt}. What is the complexity of D_{opt}? We show that D_{opt} is NP-hard. We do this by proving that we can use D_{opt} to solve an NP-hard problem. Recall that the inputs to the D_{opt} consist of a problem instance and a target processing environment. The output is the optimal decomposition of the problem. We transform an instance of a known NP-hard problem, **min-cut**, into a form suitable for input to D_{opt}: a problem instance and a target platform. We do this transformation in polynomial time. We show that the output of D_{opt} is indeed the solution to the original NP-hard problem, min-cut.

Theorem 1: D_{opt} is NP-hard.

Proof: We transform the known NP-hard problem of **minimum cut into**

bounded sets (min cut) [19] to D_{opt}.

An instance of min cut is as follows: Graph $G = (V, E)$, specified vertices $s, t \in V$, weights $w(e) = 1$ for each $e \in E$. Let $n = |V|$.

Problem: Partition V into two disjoint sets V_1 and V_2 such that $s \in V_1$ and $t \in V_2$, and the number of edges that have one end point in V_1 and one end point in V_2 is a minimum. Require $|V_1|$ and $|V_2|$ differ by at most 1.

Transformation:

Step 1 : Attach a list to each $v \in V$. Each list contains n^4 randomly chosen numbers.

Step 2: Construct a machine specification with two processing nodes of equal power and a communication link between them. Assume t_{comp} as the time for a floating point operation and t_{comm} as the time for communicating a number through the link. The values of t_{comm} and t_{comp} are specified with the following constraint:

$\frac{|V|^2}{2} t_{comp} < t_{comm} < |V|^2 t_{comp}$

No overlapping of communication and computation is permitted.

Each processor has the same amount of memory and can only store a maximum of $\lceil |V|/2 \rceil$ vertices.

Step 3: Restate the problem as: Given the graph $G = (V, E)$ modified as shown above, we need to compute at each node, other than s and t, the sum of a sequence of numbers, made up from the lists of its neighbors. Each neighbor contributes the k^{th} item in its list to the sequence, where k is the label of the requesting node. (Nodes are labeled 1 to n.) At s and t, we need to compute the sum of the numbers in its own list. Dividing this problem to minimize execution time on the parallel machine is the new problem. We call this **opt divide**.

Claim : A solution to opt divide implies a solution to min cut.

Let $V \rightarrow \{V_1, V_2\}$ be a solution to opt divide. To show this is a

solution to the min cut problem as well, we prove the following statements:

- $|V_1|$ and $|V_2|$ differ by at most 1.

  Memory constraint demands that $\max(|V_1|, |V_2|) = \lceil |V|/2 \rceil$.

  This ensures that $-1 \leq (|V_1| - |V_2|) \leq 1$.

- s and t will not be in the same subset.

  To show that any optimal solution should have s and t in different processors, we compute a lower bound on the run time of a solution that allocates them on the same processor. We use $T_{s\&t}$ to denote this run time.

$$T_{s\&t} \geq 2(n^4 - 1) \tag{2.1}$$

  This follows from the fact there are n^4 items in the list, it takes $(n^4 - 1)$ operations to find the sum, and at s and t we need to compute these sums.

  We can also compute an upper bound on the run time of a solution that allocates them on different processors. We use $T_{s|t}$ to denote this run time.

$$T_{s|t} \leq (n^4 - 1) + \frac{n}{2}n + t_{comm}\frac{n}{2}n \tag{2.2}$$

  The first term above computes the computation time at s or t, the second term is an upper bound on the computation time at the remaining nodes, and the third term is an upper bound on the communication time.

  From (2.1) and (2.2) above, it is clear that $T_{s\&t} > T_{s|t}\ \forall n \geq 2$ if $t_{comm} < n^2 t_{comp}$. Since this restriction on communication time is specified as part of the input to opt divide, it follows that any optimal solution will have s and t on different nodes.

- The number of edges that have one end point in V_1 and one end point in V_2 is a minimum.

If this is not the case, then there is a bisection of G with fewer edges crossing the cut *and* satisfying the two constraints above. Let us compute an upper bound on the run time of such a solution. We use T' to denote this run time. Let c be the number of edges crossing the cut in this solution.

$$T' \leq (n^4 - 1) + \frac{n}{2}n + ct_{comm} \tag{2.3}$$

The first term above computes the computation time at s or t, the second term is an upper bound on the computation time at the remaining nodes, and the third term is the communication time.

Next step is to compute the lower bound on the run time of the solution to opt divide, which we call T_{Dopt}.

$$T_{Dopt} \geq (n^4 - 1) + (c + 1)t_{comm} \tag{2.4}$$

The first term above computes the computation time at s or t and the second term is a lower bound on the communication time. From (2.3) and (2.4), it follows that a solution that does not minimize the edges across the cut will not be optimal as long as $t_{comm} > \frac{n^2}{2}t_{comp}$. Once again, the constraint on the communication and computation times ensures this requirement is satisfied.

Since any solution to opt divide satisfies the three constraints above, it is a solution to min cut as well. Conversely, a solution to the min cut problem implies a solution to opt divide.

Since D_{opt} does precisely what is required to solve opt divide, and since min cut is known to be an NP-hard problem, D_{opt} is also NP-hard. □

This result has practical significance in developing a system for architecture-adaptable parallel processing based on the divide-and-conquer paradigm. Since it

is not pragmatic to have a generic and optimal divide function, we focus on good heuristics to partition the problem. By restricting the system to be domain-specific, we improve our chances of designing near-optimal partitioning algorithms.

2.4.3. Benefits of Divide-and-Conquer

Several advantages result from using the divide-and-conquer model for representing the solution space. Its algebraic structure permits easy and elegant representation of the solution space and accurate performance prediction. There is a natural relation between divide-and-conquer and parallel processing, which gives us access to the structure of an algorithm. Architecture adaptability is accomplished by the manipulation of this structure. Figure 2.2 shows an example where this structure is exploited to map computations onto an arbitrary processing environment. An implementation as shown in Figure 2.2 will give good performance for a number of reasons:

- We are able to employ the best sequential algorithm as the base case at each single processor partition.
- We use only regular communication patterns, reducing the communication overhead.
- We minimize communication across machine boundaries, which is more expensive than intra-machine communication.

What is the price of these benefits? Conventional wisdom dictates we are limiting the scope of the system by using a restricted computational model. If the class of problems that can be efficiently solved using this model is too restrictive,

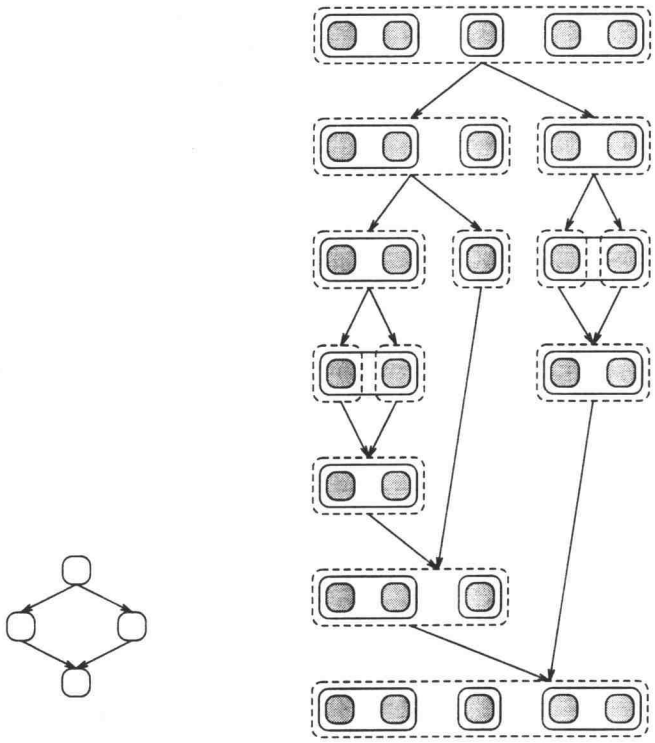

FIGURE 2.2. Mapping of the DC operation to an arbitrary processing environment. The simple divide-and-conquer operation shown on the left is mapped as shown on the right. The shaded squares denote the processors. A single-processor partition triggers the base function. The processing environment consists of two two-processor machines and a single processor machine.

then the worth of the system is questionable. Our next task is to characterize this class using parallel complexity theory.

2.4.4. Power of Divide-and-Conquer

We use the *circuit model* as the model for parallel computation [20]. A *circuit* is a DAG with ordered edges and labeled nodes, where each node has a bounded fan-in. The *size* of the circuit is the number of nodes in the DAG, and the *depth* of the circuit is the length of the longest path in the DAG.

The nodes in the DAG with zero fan-in are called the input nodes, and nodes with zero fan-out are called output nodes. Let $B = \{0, 1\}$. A boolean circuit with n input nodes and m output nodes computes a Boolean function $f : B^n \rightarrow B^m$. This notion can be generalized by having a family of circuits that solves a problem P, where P is a function from B^* to B^*.

The circuit model is an important tool for defining complexity classes. But to compute with a circuit family, we must be able to construct the circuit for each input size. A *uniform* family of circuits that solves some problem is a family of circuits for which there is a computationally simple rule to construct the various circuits [21]. There are several notions of uniformity; we will use polynomial-time uniformity in this paper [22]. A family of circuits C is polynomial-time uniform if the description of the n^{th} circuit C_n can be generated by a Turing machine in polynomial time.

Class NC and its practical significance: The class NC is defined as the set of problems that can be solved using uniform circuit families of depth bounded by a polylogarithmic function of the input size, and size bounded by a polynomial function of the input size. The class NC^k is a subclass of NC where k is the degree of the bounding polylogarithmic function [22].

The problems in the class NC have the potential to benefit from parallelization. In this section, we show that any problem in NC^k can be solved using our computational model without changing its complexity class. In other words, if there is a uniform circuit family that solves some problem in time $T(n) = O(\log^k(n))$, then there is an "algorithm" based on our model of divide-and-conquer that solves the problem in time $T(n) = O(\log^k(n))$.

BDC-DAG: a restricted circuit model. To represent the restrictions imposed by our DC model, we introduce a restricted version of DAG. We call this

BDC-DAG for Binary Divide-and-Conquer DAG. A BDC-DAG has the following characteristics:

- The nodes in a BDC-DAG can be arranged in rows, the number of rows being the same as the depth of the DAG, d.
- Each row has the same number of nodes, which is the width of the DAG, w.
- There are exactly $\log n$ rows in each BDC-DAG.

Additionally, a BDC-DAG has restrictions on the permissible edges. This is best explained by partitioning BDC-DAGs into two categories: *fan-in* BDC-DAG and *fan-out* BDC-DAG. The former maps its nodes onto a fan-in binary divide-and-conquer tree, while the latter maps them onto a fan-out DC tree. Every edge in the BDC-DAG is between adjacent levels. Permissible edges in a BDC-DAG can be grouped into two categories:

1. Straight-down edges: These edges connect nodes at the same position between adjacent levels.
2. Cross edges: Any other edge should satisfy the following constraint: if an edge connects node s at level k to the node t at level $(k+1)$, then the node s should map into the partner node of the DC tree to which node t is mapped.

Figure 2.3 shows examples of BDC-DAGs. We define our computational model based on divide-and-conquer as a concatenation of several BDC-DAGs.

Theorem 2: A uniform family of circuits that solves some problem can be reformatted as a concatenation of BDC-DAGs without changing its parallel complexity class.

Proof: The proof is by construction. We are given a uniform circuit family C

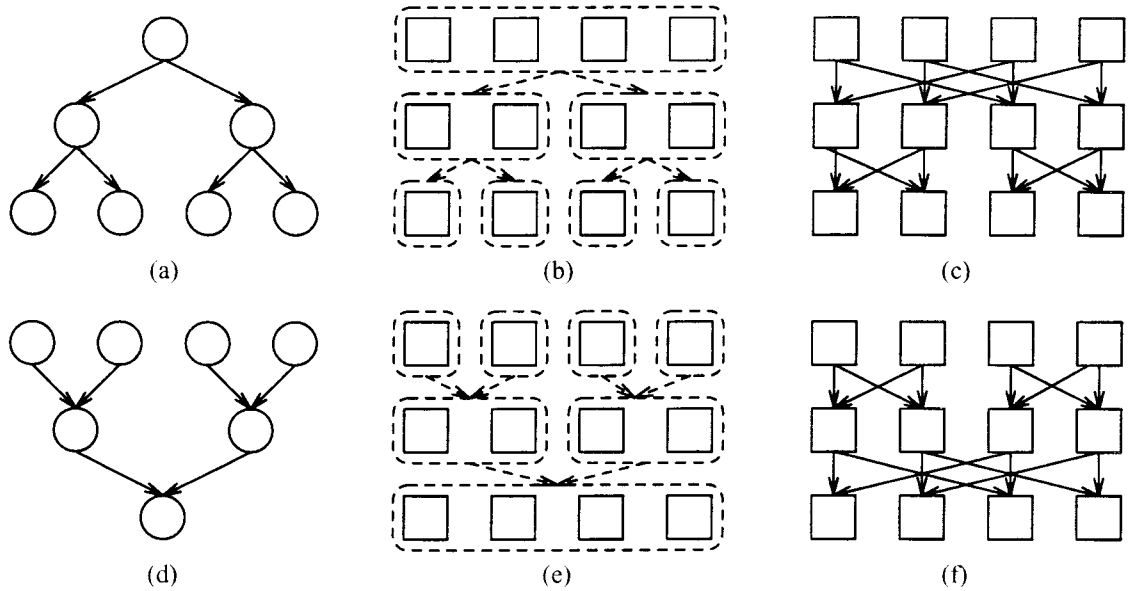

FIGURE 2.3. Examples of BDC-DAGs. (a) A fan-out DC tree. (b) The mapping of the DC tree to a four processor system. (c) An example of a fan-out BDC-DAG. (d) A fan-in DC tree. (e) The mapping of the DC tree to a four processor system. (f) An example of a fan-in BDC-DAG.

with depth $d = O(\log^k(n))$ that solves some problem P. Our goal is to construct a uniform circuit family C^{dc} with depth $d^{dc} = O(\log^k(n))$, where C^{dc} is merely a concatenation of several BDC-DAGs. n is the problem size. For each $C_i \in C$, we construct the corresponding $C_i^{dc} \in C^{dc}$.

Step 1: Convert C_i into a uniform width circuit (Figure 2.4b). This is a very simple operation, and can be accomplished with the help of a topological sort of the DAG.

Step 2: Partition C into blocks of rows where each block has at most $\log n$ rows. The number of blocks is given by $b = \lceil \frac{d}{\log n} \rceil$. All blocks, except the last one, will have exactly $\log n$ rows, with the first one having the first $\log n$ rows of the original DAG and so on.

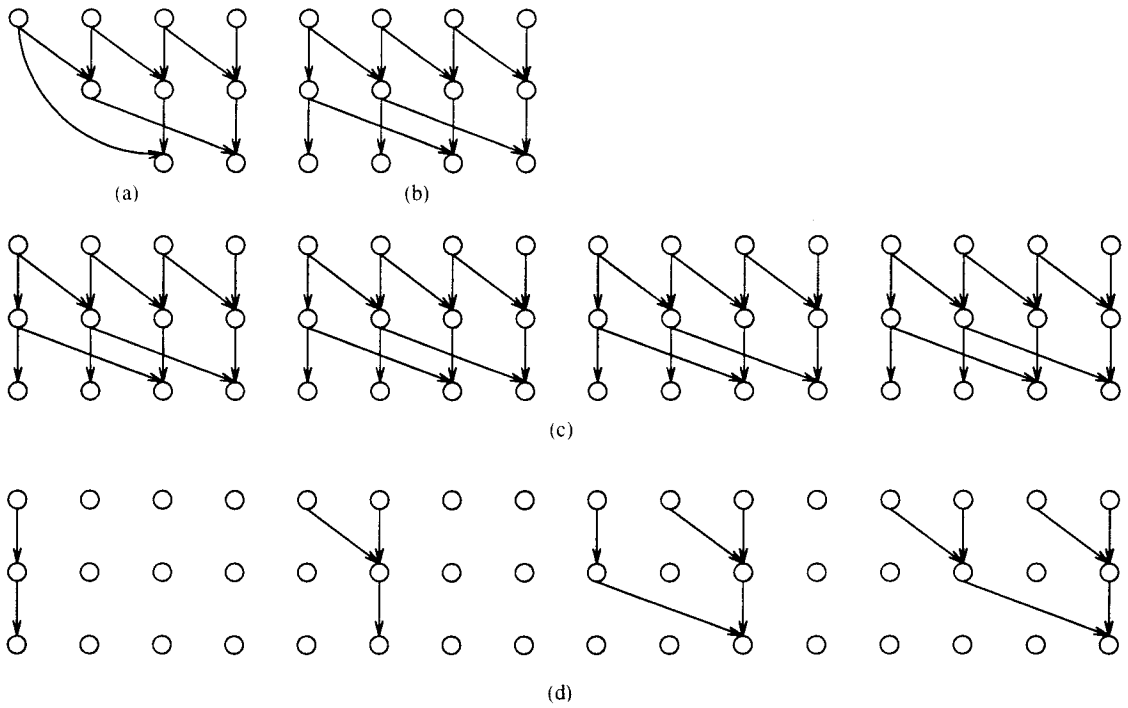

FIGURE 2.4. Construction of the BDC-DAG from an arbitrary DAG. (a) The input DAG. (b) DAG after step 1. (c) DAG after step 3. (d) DAG after step 4.

Step 3: Replicate each block n times laterally (Figure 2.4c). Now we have a 2-D grid of blocks. $Block_{i,j}$ is the j^{th} replication of the i^{th} block.

Step 4: Delete all edges from each block $Block_{i,j}$ except those which affect the computation at the j^{th} output node (Figure 2.4d). Thus, $Block_{i,j}$ is, in effect, responsible for computing only the j^{th} item of the output.

Step 5: Rearrange the nodes such that each block is a fan-in BDC-DAG. It is guaranteed that each block can be easily rearranged to form a BDC-DAG because we have bounded degree of fan-in. Without losing generality, let us assume that the fan-in is 2. Thus the output node is connected to two nodes at one level up; these two are connected to four nodes at one more level up; and so on. The structure of this DAG is that of a rooted binary tree.

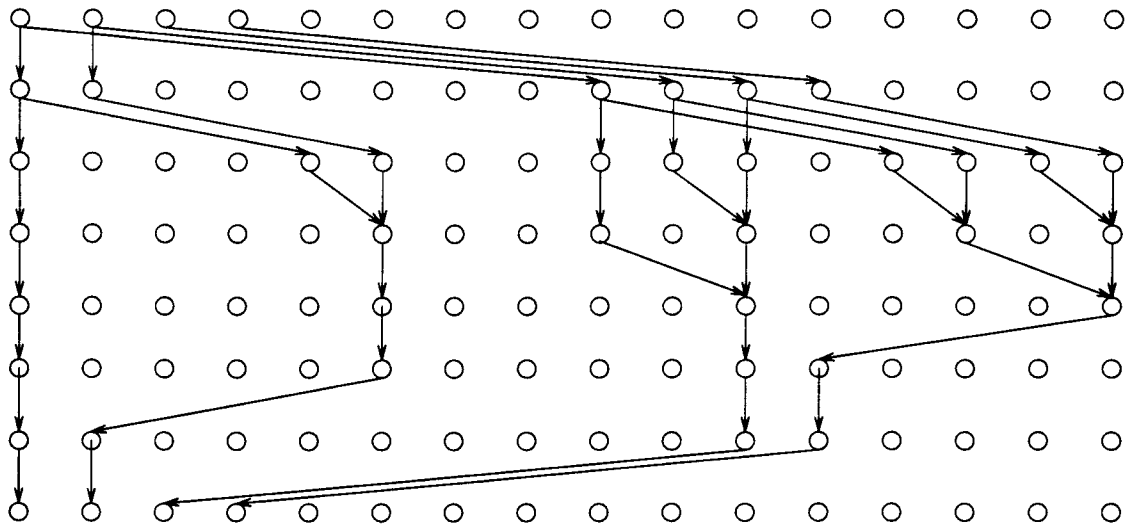

FIGURE 2.5. The final BDC-DAG constructed from the DAG in the previous Figure.

Step 6: Insert *scatter* and *gather* phases (Figure 2.5). We insert a *scatter* phase before the top of each block and append a *gather* phase to the bottom, each having a depth of $\log n$. In the scatter phase, the n inputs are broadcast to each of the n blocks. In the gather phase, the outputs generated from each block are combined.

Depth of the circuit: Now we compute the depth of the resulting circuit, d^{dc}. Each block in the constructed BDC-DAG has a depth of $3 \log n$. This gives us $d^{dc} = 3 \times (\log n) \times \frac{d}{\log n} = O(\log^k(n))$.

Size of the circuit: Since each block is replicated n times, each level of the BDC-DAG has n^2 nodes. But the size is still bounded by a polynomial function of n.

Uniformity: Since the original family of DAGs C satisfies the uniformity criterion and the construction can be done in polynomial time, the family of BDC-

DAGs we constructed, C^{dc}, is also a uniform family of circuits which solves the same problem.

Hence, we reformatted the original circuit as a concatenation of BDC-DAGs without changing its parallel complexity class. □

2.4.5. Discussion

From a theoretical perspective, this result is very important since it shows the set of problems that can be efficiently solved using our "restricted" computational model is the same as the class NC. The method of construction we used in proving the theorem has significant practical use even though it resulted in an increase in the number of nodes of the DAG. The general strategy we use to represent complex applications using our computational model is adapted from the method of construction presented above. This strategy involves breaking the computations in the application into several primitive functions and using a divide-and-conquer operation for computing each primitive function. Thus, each application is formulated as a composite function composed of several primitive functions.

Our methodology for primitive-function implementation using divide-and-conquer departs significantly from the construction given above. Thus, in reality, the increase in the number of nodes does not occur. The mechanism for implementing a primitive-function using divide-and-conquer is *ad hoc*. For example, the original DAG shown in Figure 2.4a may be thought of as computing the *scan* function. There is a very simple and elegant BDC-DAG for computing scan, as shown in Figure 2.6.

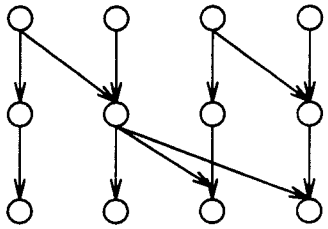

FIGURE 2.6. Scan operation represented as a BDC-DAG.

In the rest of the paper, we show the implementation of several linear algebra kernels using our computational model. We give performance benchmarks on a multiprocessor and a multicomputer.

2.5. From Theory to Practice

In this section, we describe the representation of the algorithms for four linear algebra kernels—dot product, matrix multiplication, banded-matrix vector multiplication, and tridiagonal system solver—using our computational model and the performance evaluation on the Sequent Symmetry multiprocessor and the CM-5.

2.5.1. Algorithm Representation

A higher-order function *Parallel Divide-and-Conquer (PDC)*, is used to represent the divide-and-conquer template. This function is based on the algebraic model of divide-and-conquer developed by Mou and Hudak [23]. The function PDC has 6 arguments as listed below:

1. the divide function: The domain of this function is the processor pool of the target platform. The default divide function is the *left-right divide*, in which we split the processors evenly into a left-right pair.

When a problem is mapped to a set of processors, the data structures associated with it are distributed among the processors' memories. We accomplish the division of the problem into subproblems by merely partitioning the set of processors to which it was mapped, without explicit movement of data among multiprocessor memories.

2. the pre-adjust function: This function, if present, is applied to the data in the left-right pair.

3. the base predicate: Infinite recursion is prevented using the base predicate.

4. the base function: If the base predicate returns *true*, the *base function* is applied to solve the problem directly without any further division.

5. the post-adjust function: The partial solutions resulting from solving the subproblems in the left-right pair are modified using the post-adjust function.

6. the combine function: This is simply the inverse of the divide, in which a left-right pair is merged to form the original processor pool.

The pseudocode for the PDC function is given below:

```
PDC (divide function, pre-adjust function, base predicate, base function,
        post-adjust function, combine function)
    If base predicate returns false:
        Apply divide function
        Apply pre-adjust function
        Apply PDC on both partitions
        Apply post-adjust function
        Apply combine function
    Else:
        Apply Base function
```

The first three programs - dot product, matrix multiplication, and the banded-matrix vector multiplication - employ commonly used parallel algorithms which we have merely expressed using the DC format. The last one, a tridiagonal

system solver, is adapted from [24]. We have made a significant change in the original algorithm, vastly improving the parallel performance at the expense of a certain loss in accuracy. Elsewhere, we have shown that this loss in accuracy is insignificant in real-life applications [9]. In what follows, we present these algorithms using the PDC function. Figures are included to aid in the easy understanding of the algorithmic structure. For the tridiagonal solver, the original algorithm is described along with our version.

The base predicate, in all cases presented here, simply checks for the size of the processor partition. It returns true if there is only one processor in each partition.

2.5.1.1. Dot Product

See Figure 2.7 for a graphical representation of the algorithm.

Data distribution:
    Vectors are distributed among the processors.
Divide function:
    default left-right divide
Combine function:
    default left-right combine
Pre-adjust function:
    none
Post-adjust function:
    Add results from left and right and store it on the left
Base function:
    sequential dot product routine

2.5.1.2. Matrix Multiplication ($A = BC$)

The algorithm is visually portrayed in Figure 2.8.

Data distribution:
    Matrices A and C distributed *row-wise* among the processors.

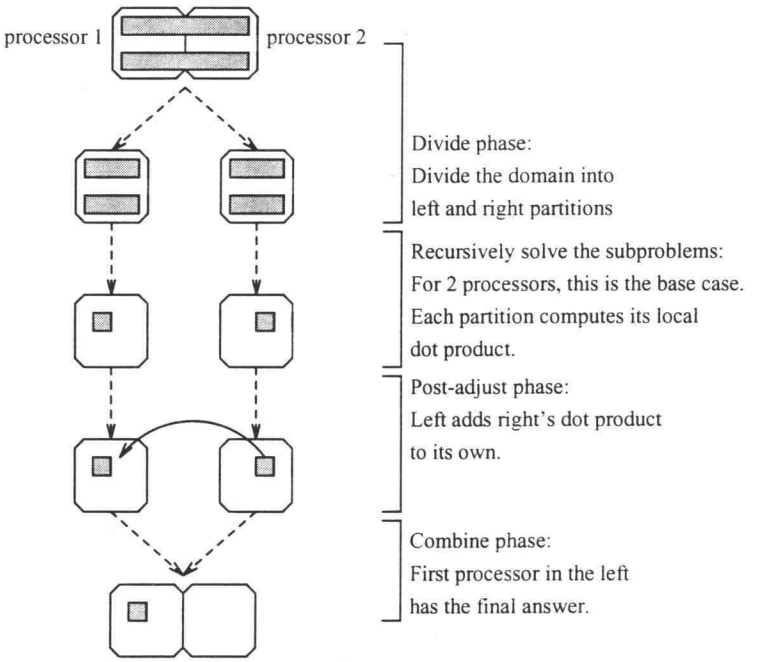

FIGURE 2.7. Algorithm for dot product executed on two processors.

Matrix B distributed *column-wise* among the processors.
Divide function:
default left-right divide
Combine function:
default left-right combine
Pre-adjust function:
none
Post-adjust function:
Swap matrix B between left and right partitions;
Invoke *Matrix Multiplication* on both partitions.
Base function:
sequential matrix multiplication routine

2.5.1.3. Banded-Matrix Vector Multiplication

Figure 2.9 shows the working of the method graphically.

Data distribution:
Matrix distributed *row-wise* among the processors.

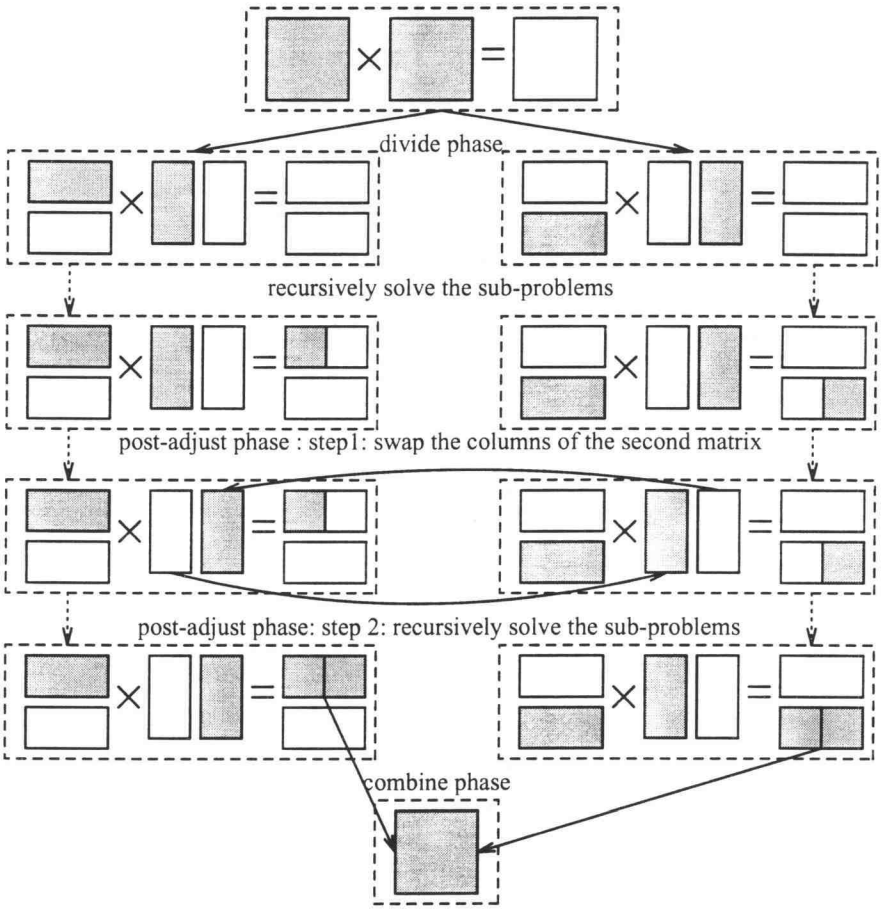

FIGURE 2.8. Schematic view of the algorithm for matrix multiplication.

Vector distributed among the processors.
Divide function:
default left-right divide
Combine function:
default left-right combine
Pre-adjust function:
Extend the right-hand vector upward by copying from the tail of the left-hand vector;
Extend the left-hand vector downward by copying from the head of the right-hand vector.
Post-adjust function:
none
Base function:
sequential banded-matrix vector multiplication routine

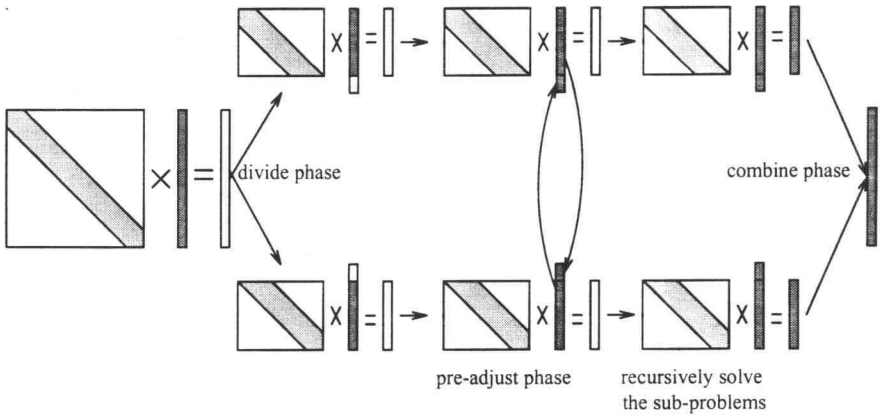

FIGURE 2.9. Schematic view of the algorithm for banded-matrix vector multiplication.

2.5.1.4. Tridiagonal System of Equations($A\mathbf{x} = \mathbf{d}$)

Original algorithm [24]:

TERMINOLOGY:

p	number of processors
n	number of unknowns
k	n/p
$\vec{e}_1$	first unit vector
$\vec{e}_k$	last unit vector
$\vec{a}$	vector forming the diagonal elements of A
$\vec{b}$	vector forming the lower diagonal elements of A
$\vec{c}$	vector forming the upper diagonal elements of A

$$A_{j+1} = \begin{pmatrix} a_{jk+1} & c_{jk+1} & & & \\ b_{jk+2} & a_{jk+2} & \ddots & & \\ & \ddots & \ddots & c_{(j+1)k-1} & \\ & & b_{(j+1)k} & a_{(j+1)k} & \end{pmatrix}$$

$$\vec{x_j} = \begin{pmatrix} x_{jk+1} \\ x_{jk+2} \\ \vdots \\ x_{(j+1)k} \end{pmatrix}$$

$$\vec{d_j} = \begin{pmatrix} d_{jk+1} \\ d_{jk+2} \\ \vdots \\ d_{(j+1)k} \end{pmatrix}$$

STEP 1:

Solve in parallel the linear tridiagonal systems of size $k = n/p$:

$$(2.5)\quad A_i\vec{y_i} = \vec{d_i} \quad \forall i = 2, 3, \ldots, p-1$$

$$(2.6)\quad A_1\vec{z_1} = \vec{e_k}$$

$$(2.7)\quad A_i\vec{z_m} = \vec{e_1} \quad \forall i = 2, 3, \ldots, p-1; m = 2i-2$$

$$(2.8)\quad A_i\vec{z_l} = \vec{e_k} \quad \forall i = 2, 3, \ldots, p-1; l = 2i-1$$

$$(2.9)\quad A_p\vec{z_{2p-2}} = \vec{e_1}$$

STEP 2:

Solve the tridiagonal system given by:

$$(2.10)\quad \begin{pmatrix} s_1 & t_1 & & \\ r_2 & s_2 & \ddots & \\ & \ddots & \ddots & t_{2p-3} \\ & & r_{2p-2} & s_{2p-2} \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{2p-2} \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{2p-2} \end{pmatrix}$$

where:

$$(2.11)\quad s_i = \begin{cases} (\vec{e}_k^T \vec{z}_i) & i = 1, 3, \ldots, 2p-1 \\ (\vec{e}_1^T \vec{z}_i) & i = 2, 4, \ldots, 2p-2 \end{cases}$$

$$(2.12)\quad r_i = \begin{cases} (\vec{e}_k^T \vec{z}_{i-1}) & i = 1, 3, \ldots, 2p-1 \\ 1/c_{\frac{i}{2}k} & i = 2, 4, \ldots, 2p-2 \end{cases}$$

$$(2.13)\quad t_i = \begin{cases} 1/b_{\frac{i+1}{2}k+1} & i = 1, 3, \ldots, 2p-1 \\ (\vec{e}_1^T \vec{z}_{i+1}) & i = 2, 4, \ldots, 2p-2 \end{cases}$$

$$(2.14)\quad \beta_i = \begin{cases} -(\vec{e}_k^T \vec{y}_{\frac{i+1}{2}}) & i = 1, 3, \ldots, 2p-1 \\ -(\vec{e}_1^T \vec{y}_{\frac{i+2}{2}}) & i = 2, 4, \ldots, 2p-2 \end{cases}$$

STEP 3:

Compute the solution $\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{pmatrix}$ using:

$$(2.15)\quad \vec{x}_i = \begin{cases} \vec{y}_1 + \alpha_1 \vec{z}_1 & i = 1 \\ \vec{y}_i + \alpha_{2i-2}\vec{z}_{2i-2} + \alpha_{2i-1}\vec{z}_{2i-1} & 2 \le i \le p-1 \\ \vec{y}_p + \alpha_{2p-2}\vec{z}_{2p-2} & i = p \end{cases}$$

An Optimization: Consider a tridiagonal system of the form:

$$(2.16)\quad \begin{pmatrix} a_1 & b_1 & & \\ c_2 & a_2 & \ddots & \\ & \ddots & \ddots & c_{n-1} \\ & & c_n & a_n \end{pmatrix} \begin{pmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vdots \\ \vec{x}_n \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

The solution vector **x** could be approximated by solving a smaller system of size $m < n$ as shown below:

$$(2.17)\quad \begin{pmatrix} a_1 & b_1 & & \\ c_2 & a_2 & \ddots & \\ & \ddots & \ddots & c_{m-1} \\ & & c_m & a_m \end{pmatrix} \begin{pmatrix} \tilde{x}_1 \\ \tilde{x}_2 \\ \vdots \\ \tilde{x}_m \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

and setting

$$(2.18)\quad \vec{x}_i = \begin{cases} \tilde{x}_i & i \le m \\ 0 & i > m \end{cases}$$

Equations 2, 3, 4, and 5 in STEP 1 of the original algorithm can be approximated using this technique. To complete the derivation of the new algorithm, we

set $p = 2$ and recursively apply the approximated solution scheme to the two partitions. If m chosen above is *less than* k, this yields a highly communication and computation efficient algorithm. (We have shown in [9] that this indeed is the case in many real-life situations.)

New algorithm: (figure 2.10)

Data distribution:
- Matrix distributed *row-wise* among the processors.
- Vectors distributed among the processors.

Terminology:
- $\mathbf{e}_{last}$ is the vector with 1 as the last component and 0 elsewhere.
- $\mathbf{e}_{first}$ is the vector with 1 as the first component and 0 elsewhere.
- $\mathbf{a}, \mathbf{b}, \mathbf{c}$ are the non-zero diagonals of A.

Divide function:
- Default left-right divide.
- A is broken into A_l and A_r, which are the top-left and bottom-right quadrants of A respectively.

Combine function:
- Default left-right combine

Pre-adjust function:
- None

Post-adjust function:
- Left-side:
  - Solve for $\mathbf{z}_l$ using $A_l\mathbf{z}_l = \mathbf{e}_{last}$;
  - Copy from Right : $\beta = \mathbf{x}_r[0]; gamma = \mathbf{z}_r[0]; \delta = \mathbf{a}_r[0]$;
  - Compute α as a function of $\beta, \gamma, \delta, \mathbf{x}_l[n-1], \mathbf{z}_l[n-1]$, and $\mathbf{c}_l[n-1]$;
  - Update $\mathbf{x}_l$ as $\mathbf{x}_l + \alpha\mathbf{z}_l$.
- Right-side:
  - Solve for $\mathbf{z}_r$ using $A_r\mathbf{z}_r = \mathbf{e}_{first}$;
  - Copy from Left : $\beta = \mathbf{x}_l[n-1]; \gamma = \mathbf{z}_l[n-1]; \delta = \mathbf{c}_l[n-1]$;
  - Compute α as a function of $\beta, \gamma, \delta, \mathbf{x}_r[0], \mathbf{z}_r[0]$, and $\mathbf{a}_r[0]$;
  - Update $\mathbf{x}_r$ as $\mathbf{x}_r + \alpha\mathbf{z}_r$.

Base function:
- Sequential solver for tridiagonal system

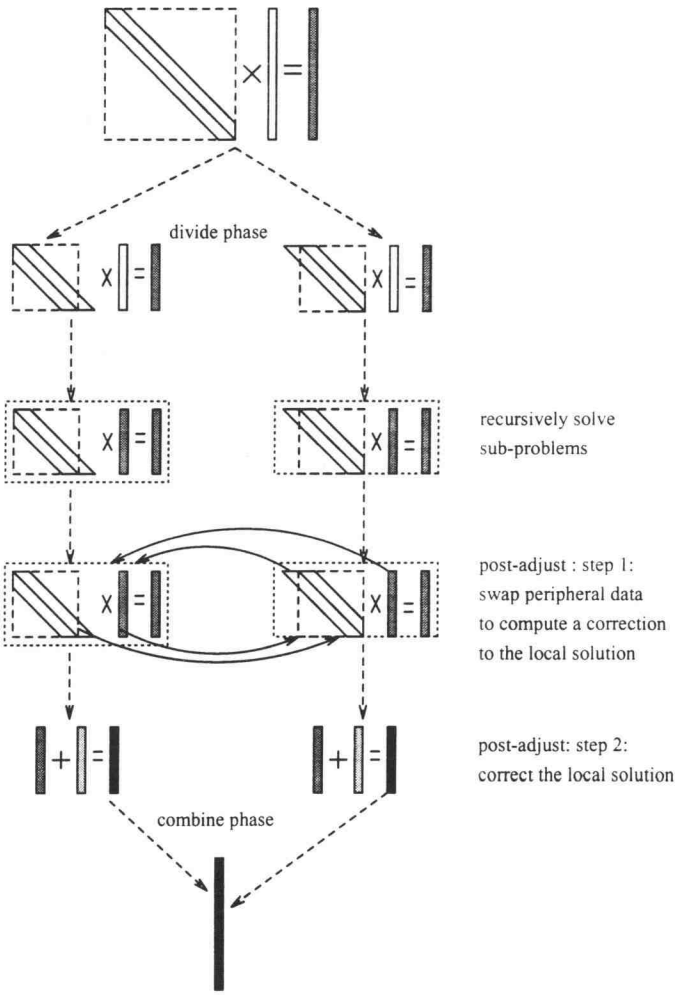

FIGURE 2.10. Schematic view of the algorithm for tridiagonal system solution.

2.5.2. Performance Evaluation

We have implemented this system on the Sequent Symmetry, a shared-memory multiprocessor, and Thinking Machines CM-5, a distributed-memory multicomputer [25]. On the Sequent Symmetry, we implemented the system using Dataparallel C, a SIMD superset of the C programming language targeted for MIMD computers [2]. This is a high-level parallel programming language featuring virtual

	DOT		MATMUL		BANDED		TRIDIAG	
procs	Symmetry	CM-5	Symmetry	CM-5	Symmetry	CM-5	Symmetry	CM-5
1	1	1	1	1	1	1	1	1
2	1.88	2	1.68	2	1.93	1.98	1.47	1.89
4	3.92	3.94	3.59	3.95	3.87	3.74	2.95	3.60
8	7.83	11.8	7.04	7.8	7.25	8.08	5.83	7.30
16	11.75	23.64	13.40	14.4	11.6	15.54	10.68	17.40
32		31.52		26.2		28.9		30.67
64		47.30		41.1		50.5		48

TABLE 2.1. Speedup on Sequent Symmetry and CM-5 for *dot product*, *matrix multiplication* , *banded-matrix vector multiplication*, and *tridiagonal solver*, implemented using the divide-and-conquer template.

processors, global name space, and synchronous execution of a single instruction stream. We have no use for the virtual processor emulation capability of the language, since the structure of the divide-and-conquer tree is determined by the number of physical processors, not by the size of the data set. Nevertheless, the global name space and synchronous execution make the programming much easier.

One drawback of using a high-level language can be a loss of efficiency. This is not significant in the case of Dataparallel C on the Symmetry. On the other hand, this overhead could seriously affect the performance on a distributed-memory machine. The main source of this overhead is the inability of the language to detect the communication patterns of the divide-and-conquer algorithm, which have highly efficient implementations on many architectures, most notably on hypercubes and fat-trees.

As a consequence, we have used C with message-passing for implementation on CM-5. Programs are written in an SPMD (Single Program Multiple Data) style with calls to CMMD, a message-passing library [26].

For all the algorithms given in this paper, the base case is simply the best sequential program to solve the problem. This implies that *parallelizability* and *speedup* are one and the same for these applications [25]. Thus the speedup is computed by running the program on a single processor, which translates to a single execution of the base case. Table 2.1 shows the speedup of the divide-and-conquer algorithms on Sequent Symmetry and CM-5. For dot product, the vector length is 32,768. For matrix multiplication, matrix size is 128×128. For banded-matrix vector multiplication, matrix size is 2048×2048 with a bandwidth of 33. For tridiagonal solver, the size of the system being solved is 65,536.

Sequent Symmetry: At the maximum machine size of 16 processors, the overhead associated with the high-level language tends to lower the performance slightly, but the overall performance is attractive.

CM-5: Since we were able to implement the communication patterns efficiently using the virtual channels provided by CMMD, the performance is spectacular. As more processors are added, the impact of the collective cache memory provides superlinear speedup. This trend changes as the number of processors cross a certain threshold and the communication overhead begins to weigh in more heavily.

2.6. Related Work

Aho et. al. give the first comprehensive description and analysis of divide-and-conquer as a problem solving methodology [27]. Application of this strategy

for parallel processing was introduced in the seminal work [28]. A thorough formal treatment of the parallelism in divide-and-conquer algorithms can be found in [23].

A simple language based on C for expressing divide-and-conquer computations and an implementation using Chare Kernel is presented in [29]. DIVACON, a functional programming language based on divide-and-conquer for parallel programming, is introduced in [30]. A synthesis of object-oriented and divide-and-conquer paradigms for parallel processing is the theme of [31]. Several researchers have previously analyzed the parallel performance of divide-and-conquer [32–35]. But our approach to parallel divide-and-conquer, as outlined in this paper, is different from any previous work. Our innovative methodology enables us to use divide-and-conquer as a powerful tool, enabling architecture-adaptable, efficient, and easy-to-use parallel processing.

2.7. Epilogue

In this paper, we introduced the theoretical foundations of a methodology for architecture-adaptable parallel processing. At the heart of the methodology is a computational model based on divide-and-conquer. By using an innovative approach to implementing parallel programs using the divide-and-conquer paradigm, we showed good performance can be obtained on both shared-memory and distributed-memory platforms.

In two related papers, we have discussed various facets of this methodology in detail:

- In [8], we develop parameterized base templates and composite templates to implement the primitive and composite functions respectively. Real-world applications are represented using composite-templates which contain several

base templates. The base templates represent solution methodologies for the primitive functions. Several base templates may exist to compute a primitive function, each one leading to a different algorithm. The architecture adaptability of the methodology is displayed by mapping these templates to diverse processing environments. We use analytical performance prediction to search the solution space for the best algorithm.

- In [15], we describe the use of this methodology to build an architecture-adaptable problem solving environment for scientific computing. We show how object technology can be used to design and implement the system. We demonstrate the expressibility, efficiency, and scope of the system by automatically mapping complex applications (finite element model, Kalman Filter) onto diverse architectures (multicomputers, multiprocessors, workstation networks, SMP clusters).

3. METHODOLOGY AND PROOF OF CONCEPT

Towards Architecture-Adaptable Parallel Programming

Santhosh Kumaran and Michael J. Quinn

Scientific Programming (in press)
John Wiley & Sons, Inc.
New York, NY

3.1. Abstract

Parallel processing is facing a software crisis. The primary reasons for this crisis are the short life span and small installation base of parallel architectures. In this paper, we propose a solution to this problem in the form of an architecture-adaptable programming environment. Our method is different from high-level procedural programming languages in two ways: (1) our system automatically selects the appropriate parallel algorithm to solve the given problem efficiently on the specified architecture; (2) by using a divide-and-conquer template as the basic mechanism for achieving parallelism, we considerably simplify the implementation of the system on a new platform. There is a trade-off, however: the loss of generality. From a pragmatic point of view, this is not a major liability since our strategy will be useful in building domain-specific problem solving environments and application-oriented compilers, which can be easily and *effectively* ported to diverse architectures. We give preliminary results from a case study in which our method is used to adapt the parallel implementations of the conjugate gradient algorithm on a multiprocessor, a multicomputer, and a workstation network.

KEY WORDS: Architecture adaptability; parallel divide-and-conquer; templates; performance prediction; parallel programming environments; scientific computing.

3.2. Introduction

The most efficient parallel algorithm for solving a problem often depends on the target architecture. Thus, unless a parallel programming system has the ability to adapt the algorithm to the architecture, it will not be truly machine-independent.

In the traditional approaches to machine-independent parallel programming, the user encodes an algorithm as a parallel program using a high-level programming language. Using a combination of compilers and run-time systems, this program can be executed on a variety of platforms, but the algorithm embedded in the program may not execute efficiently on all the platforms. Hence only limited machine independence is achieved.

In this paper, we present a new scheme for machine-independent parallel programming. Our scheme is built on the following three key ideas: (1) the use of a database of parameterized algorithmic templates to represent *computable* functions; (2) frame-based representation of processing environments; and (3) the use of an analytical performance prediction tool for automatic algorithm design.

By automating the detailed design of an algorithm and the generation of a parallel program, our approach relieves the user from much of the burden of parallel programming. There is a trade-off, however: the set of problems that can be solved efficiently using our approach is limited by the contents of the template database. However, we believe our strategy will be useful in building domain-specific *problem solving environments* and *application-oriented compilers*, which can be easily and *effectively* ported to diverse architectures.

Figure 3.1 contrasts our approach with the traditional approach. In our problem-oriented approach, the user describes the problem to be solved, rather than an algorithm to solve the problem. The set of problems that can be solved using a

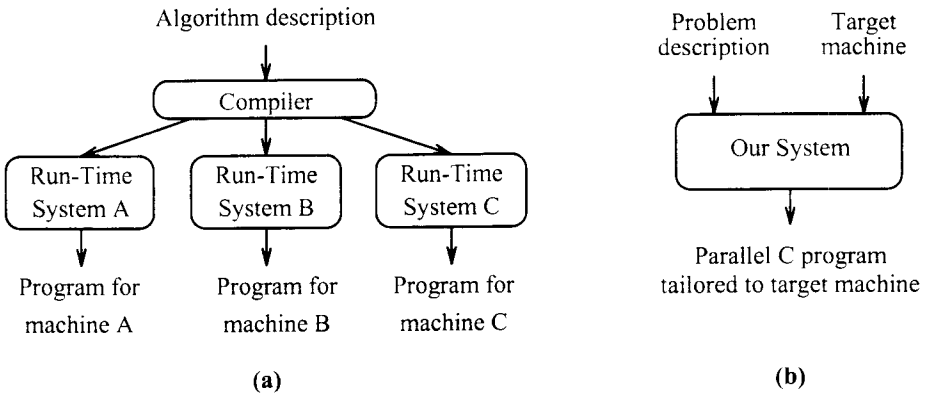

FIGURE 3.1. (a) Traditional algorithm-oriented approach to parallel processing. (b) Our problem-oriented approach to parallel processing.

system may be called the *scope* of the system. We restrict the scope of our system to provide a portable, easy to use, and high performance processing environment. In contrast, the traditional approach maximizes the scope to include all *Turing computable* problems at the expense of restricting portability, programmability, and performance.

To see that limited scope is not a major liability, one only needs to look at the recent history of the computer industry.

- The massive surge in the popularity of personal computers is primarily due to the availability of domain-specific software packages with restricted scope, prime examples being word processors and spreadsheets.

- In the realm of scientific computing, users are increasingly moving towards *Problem Solving Environments* such as MATLAB, abandoning the traditional programming languages, such as Fortran.

- The biggest challenge to parallel computing comes from the "killer workstations" simply because the improvement in performance resulting from using

the parallel computer is not enough to justify the additional cost and effort. Pragmatically, this implies that it is fruitless to parallelize *all* applications. Those that benefit from parallelization form a subset, and programming models with enough expressive power to cover a reasonable number of applications will have just as much practical use as a Turing equivalent model.

We use a computational model based on divide-and-conquer to design the algorithm templates. In the next few sections, we describe this model and the details of our scheme to automatically generate architecture-adaptable parallel programs. We have applied our scheme to develop efficient parallel programs for several scientific applications on diverse architectures. Included in this paper is a case study describing the application of our strategy to parallelize the conjugate gradient method on a shared memory multiprocessor, a distributed memory multicomputer, and a network of workstations. We believe the diversity of the target platforms and the complexity of the application make this case study a good test of the validity of our approach.

3.3. Computational Model

There is only one basic mechanism for parallelism in our model: a meta-function called *parallel divide-and-conquer*. Divide-and-conquer is a well-known problem-solving strategy in which a problem is solved by dividing it into a number of smaller subproblems and then solving the subproblems by the recursive application of the same procedure. Infinite recursion is prevented by using a *base predicate* which triggers a *base function*. The solutions to the subproblems form partial results, which are combined to form the final result. In parallel divide-and-conquer,

the subproblems are solved in parallel, providing an easy opportunity for exploiting parallelism in architecture.

In our model, a program is represented as a sequence of divide-and-conquer operations. Figure 3.2a shows the graphical representation of such a program in the form of a DAG, comprising three divide-and-conquer operations. The shaded squares denote the base cases. Note that the number of subprograms generated and the depth of recursion change for each invocation of the operation. Essentially, each operation has a well-defined top-level structure, but the details can change for each invocation. We use the notion of a *parameterized template* to represent these operations; the template describes the top-level structure and the parameters are used to add the details. The lowest layer of our template database is made up of such templates. Meta-templates, consisting of cascaded divide-and-conquer operations such as the one shown in Figure 3.2a, are formed from these *base* templates.

Another important aspect of our computational model is the mapping of the subproblems to the processing nodes. We combine the divide-and-conquer paradigm with the Single Program Multiple Data (SPMD) style of programming to obtain an efficient implementation of the cascaded divide-and-conquer explained above. The subproblems at each level of the DAG are mapped to all or a subset of the processors. This is in contrast to the conventional approach to divide-and-conquer programming where each subproblem gets mapped to a single processor. Figure 3.2b shows a possible implementation of the program in Figure 3.2a using two processors for the first divide-and-conquer operation and four processors for the rest of the computation.

A meta-template is an abstract, high-level representation of a generic method to solve a problem. There may be a large number of plausible implementations for a meta-template. We generate an efficient program to solve a problem on a given

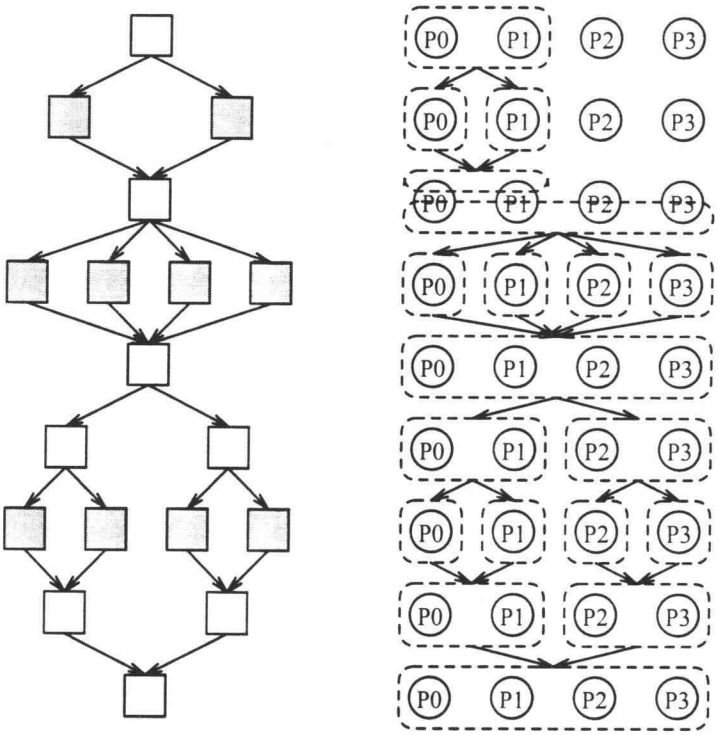

FIGURE 3.2. Mapping of the DC-tree to the processing nodes. Task graph shown on the left is mapped as shown on the right. The shaded squares denote the base cases. Note that the first DC-operation is performed using only two processors while the rest of the computation uses four processors.

architecture by choosing the implementation that performs best on that architecture. Thus we see that there is a search space associated with each meta-template and the problem of generating an efficient program reduces to a search problem. The size of this search space is determined by the number of constituent base templates, the number of parameters in each one of them, and the number of permissible values for each of these parameters. Note that there could be several meta-templates to solve the same problem, adding one more dimension to the search space.

3.4. Methodology

We begin with a collection of meta-templates for the problem and an abstract description of the architecture. The templates represent *methods* for solving the problem. The number of templates in the collection is problem-dependent—some problems will have only a single template, while others may have two or more. Our goal is to generate an efficient algorithm to solve the problem on the specified architecture.

To achieve this goal, we traverse the path from a generic method to an algorithm by adding the necessary details. This means customizing the template by determining the appropriate values for the parameters. If the search space is small, we can exhaustively search for the best set of values for the parameters, provided we have a good objective function. The role of the analytical performance prediction tool is to provide this objective function. Given a set of parameter values and the relevant specifications of a target platform, the tool predicts the performance of the implementation on the specified platform.

What kind of details do we need to add to the template to make it an efficient algorithm? Here is a partial list:

- Structure of the divide-and-conquer tree: This will vary based on the processing environment for the same template.
- Mapping of the processing nodes to the leaves of the tree: The mapping that minimizes the communication overhead is desired.
- Depth of the tree: This determines the granularity of the resulting parallel program.

- Optimal subset mapping: Sometimes performance may be enhanced by using only a subset of the resources.

- Machine-specific data decomposition: There are several ways grid data can be decomposed, and based on the problem instance and the architecture, a particular decomposition may be superior.

- Machine-specific solution method: When there are several candidate templates, the one that maximizes the performance needs to be selected.

A combinatorial explosion of the search space is conceivable for complex applications, making exhaustive search impractical. For these cases, there are two ways to prune the search tree:

1. We can make use of application-specific knowledge to limit the number of parameters and their permissible values. This kind of pruning is done manually at the template design stage. We will show an example of this in the case study.

2. We can use branch-and-bound algorithms to eliminate fruitless searching of unproductive branches. The system can automatically perform this pruning while searching for the best implementation.

If there is more than one template for a problem, then each one of them will be customized and the best one selected using the performance prediction tool.

Converting the detailed template to a message-passing program or a shared-memory program can be accomplished using current compiler technology [2].

Figure 3.3 shows the method schematically.

It is important to note that divide-and-conquer *is being used in this system merely as a methodology for designing the templates. Users do not write divide-and-conquer functions—they call higher-level functions like matrix-vector multiply or dot*

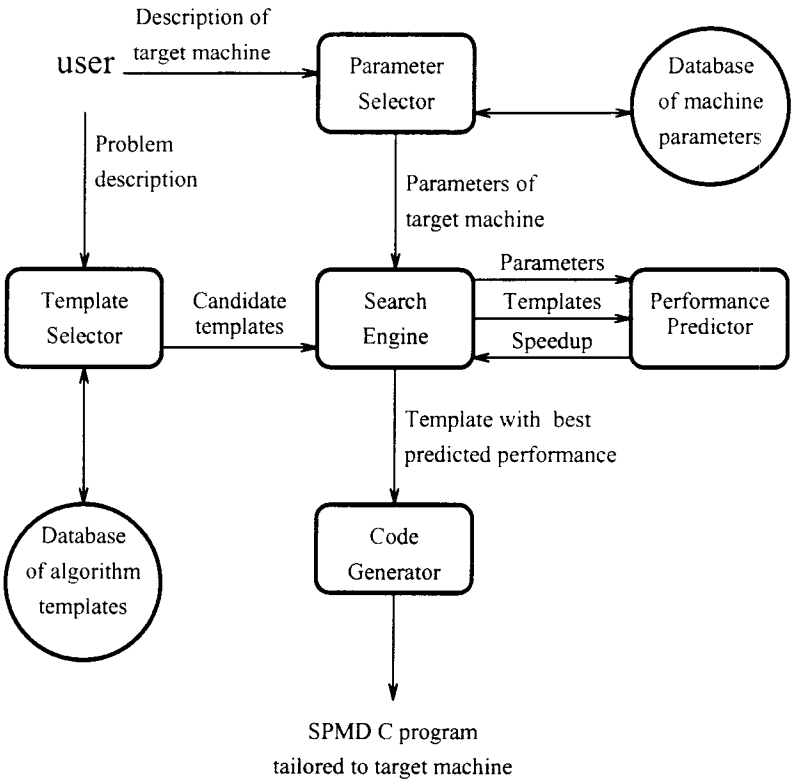

FIGURE 3.3. Schematic representation of our method for generating efficient parallel programs to solve a given problem on a specified architecture.

product. Emitted code is not a divide-and-conquer program. It is an SPMD program in which every processor is active throughout the execution of the program and doing useful work.

3.4.1. Divide-and-Conquer Template

Our template is based on the algebraic model of divide-and-conquer proposed by Mou and Hudak in [23]. The template encapsulates problem solving using divide-and-conquer in three phases: a divide phase, a conquer phase, and a combine phase. An overview of the template is given below.

- **Data distribution declarations:** This explains how data points are distributed among the processing units for distributed memory machines; for shared-memory systems, this represents the logical division of data points among processing nodes. Using grid problems as an example, row decomposition, column decomposition, and block decomposition can all be captured using appropriate data distribution declarations. These declarations can also be thought of as *pre-conditions* and *post-conditions* on the template. The distribution of input data is a necessary pre-condition for the invocation of the template; the result of the invocation (post-condition) is the output data distributed in the specified layout.

- **Divide phase:** Actions in this phase can be expressed using two functions:

  1. Divide function: As shown in Figure 3.2, the SPMD implementation of the parallel divide-and-conquer requires a mapping from the subproblems to the processing nodes. The purpose of the divide function is to specify this mapping. For example, consider the first divide-and-conquer operation in Figure 3.2. The original problem is mapped to a pool of two processors. In the divide phase, two subproblems are generated. To map these subproblems to the processing nodes, we split the processing nodes into two partitions: a left partition and a right partition. The first subproblem is allocated to the left partition and the second problem to the right partition. This is an example of a frequently-used simple divide function: binary, equal, one-dimensional, *left-right* division. Thus the domain of the divide function is the set of processing nodes of the target environment.

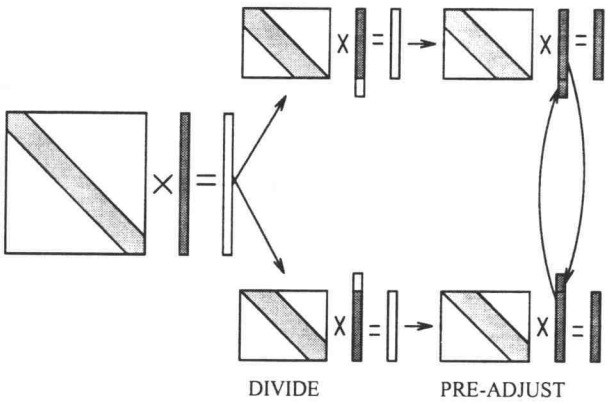

FIGURE 3.4. Divide phase of a template for banded-matrix vector multiplication. Pre-adjust function is used to extend the top half of the vector towards the bottom and the bottom half towards the top. Actual multiplication occurs inside the base function.

2. Pre-adjust function: Notice that the divide function implies the partitioning of the domain of the function computed by the template as well. For example, if the function is computing the product of a banded matrix and a vector, the left-right divide will cut the input vector and the matrix into two chunks as shown in Figure 3.4. In addition to partitioning data, the divide phase may need to modify the partitioned data sets. This is accomplished using a pre-adjust function which is applied to the partitions *before* the subproblems contained in these partitions can be solved. The divide phase of the banded-matrix vector multiplication template is illustrated schematically in Figure 3.4, where we show how the divide function splits the data sets and the pre-adjust function modifies them.

- **Conquer phase:** We need to specify only a sequential base function and a base predicate in this phase, since everything else reduces to recursive applications of the previously defined template. The base predicate is used to specify the terminating condition of the recursion. Since we use the divide function

to partition the set of processing nodes, the recursion will have to terminate when there is only a single node in a partition. Thus the default base predicate checks the number of nodes in the partition and returns true if there is only one. In this case, the base function is simply a sequential program.

In some architectures, it might be advantageous to terminate the recursion early with a group of processing nodes in each partition, instead of single node partitions. The base function will be a parallel program in such cases, but less complex than a program designed to run efficiently on the entire platform. An interesting special case is when the base functions are data-parallel programs, giving *nested data-parallelism.*

- **Combine phase:** When the subproblems are solved, we will have partial results distributed among the processors. In the combine phase, we wish to combine them to produce the final answer. Similar to the divide phase, we use two functions to accomplish this:

  1. Post-adjust function: Subproblem solutions are modified using this function. We can use the power of recursion and invoke the template itself from within the post-adjust function, if necessary. The example template for matrix multiplication, given below, illustrates this. A simpler example will be the computation of the dot product of two vectors. In the combine phase, we use the post-adjust function to add the partial results to form the global sum.

  2. Combine function: The combine function is merely the inverse of the divide function. As an example, the *left-right combine* merges the left and right partitions into a single partition.

Notice that the divide and combine functions do not operate directly on the application data. These functions merely change the state of a set of registers—which we call system data—maintained by each processor. The system data collectively determine the *position* of a processor within the DAG representing the divide-and-conquer operation. For example, consider how the position of the processor **P1** changes during the last divide-and-conquer operation in Figure 3.2. The first application of the divide function changes the system data of **P1** to make it the second processor in the left partition. The second divide makes it the first processor in the right partition of the first *left-right* pair. During the combine phase, the first application of the combine function makes it once again the second processor in the left partition. The adjust functions, which operate directly on the application data, use the position information to determine the exact nature of communications and computations at each processor.

Figure 3.5a shows the execution of the generic parallel divide-and-conquer (PDC) on n processors. Figure 3.5b shows the execution on a single processor, the base case.

Figure 3.6 shows an example template for matrix multiplication, $C = AB$. This is one of three templates we have for matrix multiplication in the system database.

Figure 3.7a shows the execution of the matrix multiplication template on n processors. The unrolled execution sequence for four processors is shown in Figure 3.7b. Figure 3.8 shows how the data structures at each processor change as execution proceeds.

Our current design of the system uses a higher-order function to implement the template with the arguments of this function representing the fields of the template. All communications appear exclusively in the adjust functions. Additionally,

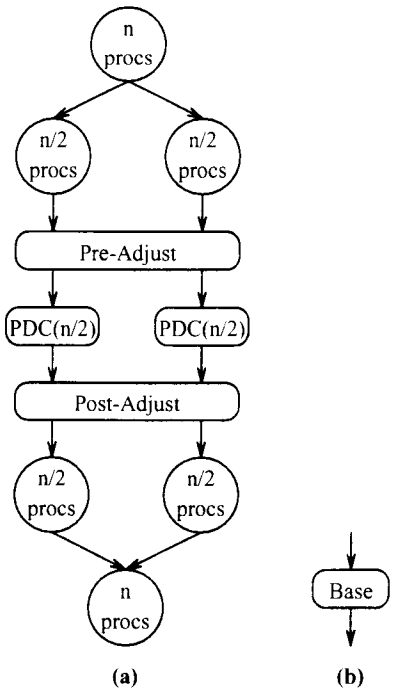

FIGURE 3.5. Schematic view of the execution of the generic parallel divide-and-conquer template (PDC). (a) PDC on n processors. (b) PDC on one processor: the base case.

these templates have the benefit of having only regular and well-defined communication patterns.

3.4.2. Representation of the Processing Environment

The computing environment is described using a frame structure. The slots in the frame represent attributes, values of which may be represented by other frames. The collection of frames, thus formed, holds all the information we need to design a program that will execute efficiently on the represented environment. The information stored in the frame includes the number of processors, the processing power of the nodes, the inter-connection network, and the memory hierarchy.

Data distribution:
matrix A distributed row-wise among the processors
matrix B distributed column-wise among the processors
matrix C distributed row-wise among the processors

Divide function:
Divide the processor pool into two equal partitions,
a LEFT partition and a RIGHT partition.
(In distributed-memory machines, this would automatically
imply the division of data structures as well.)

Pre-adjust function:
None.

Base function:
Sequential Matrix Multiplication.

Post-adjust function:
Swap columns of B between partitions.
Apply Matrix Multiplication Template to both partitions.

Combine function:
Combine the LEFT and RIGHT partitions.

FIGURE 3.6. An example template for matrix multiplication.

Figure 3.9 shows the frame representation of a typical high-performance computing environment.

3.4.3. Performance Prediction

Performance prediction plays an important role in the development of a detailed algorithm from a generic template, as pointed out in section 3. Our analytical performance prediction tool is built on the model developed by Clement and Quinn [36]. It exploits the algebraic structure of divide-and-conquer algorithms to estimate their run time on the specified processing environment.

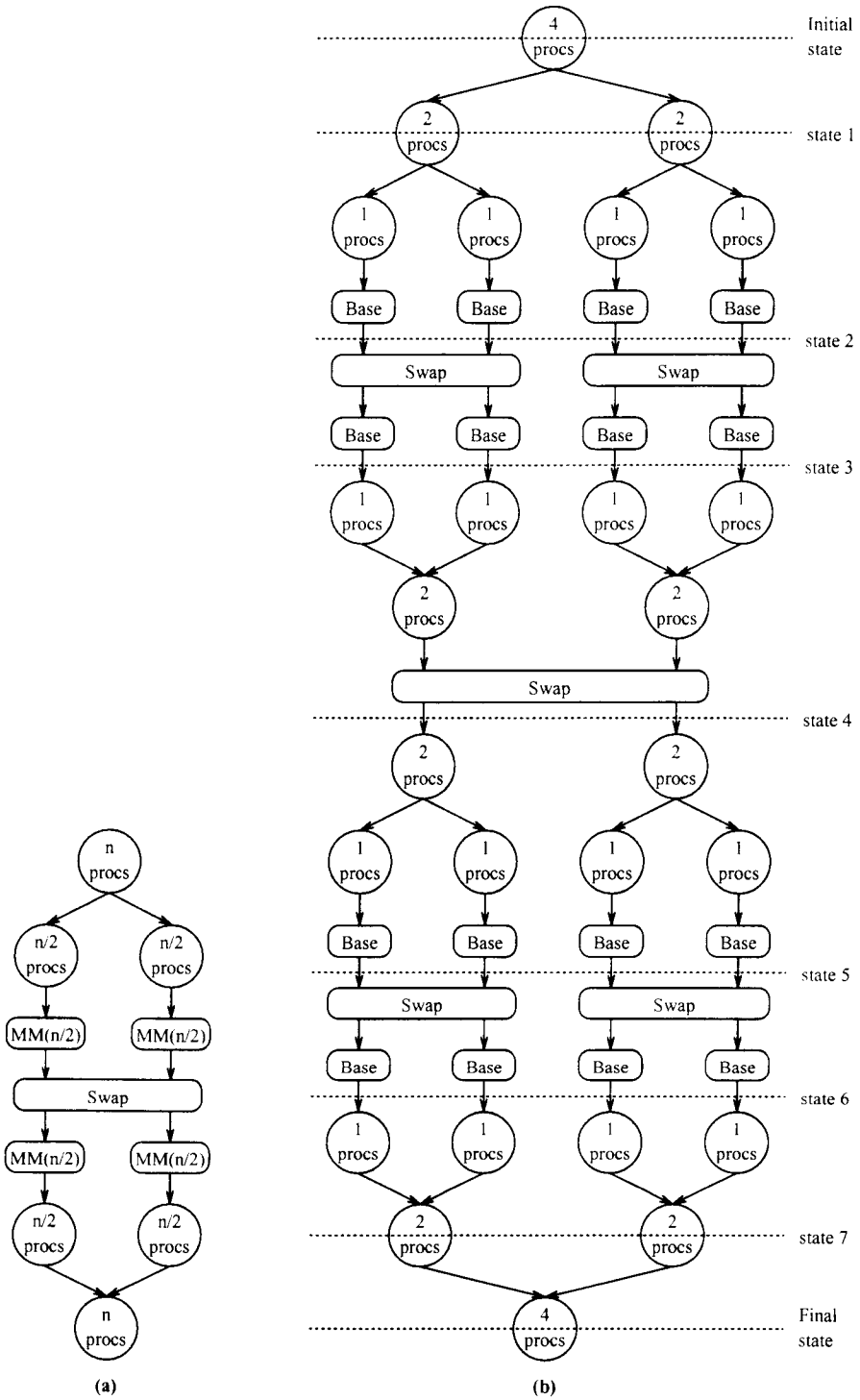

FIGURE 3.7. Schematic view of the execution of the matrix multiplication template. (a) Matrix multiplication on n processors. (b) Matrix multiplication on four processors after unrolling the recursion.

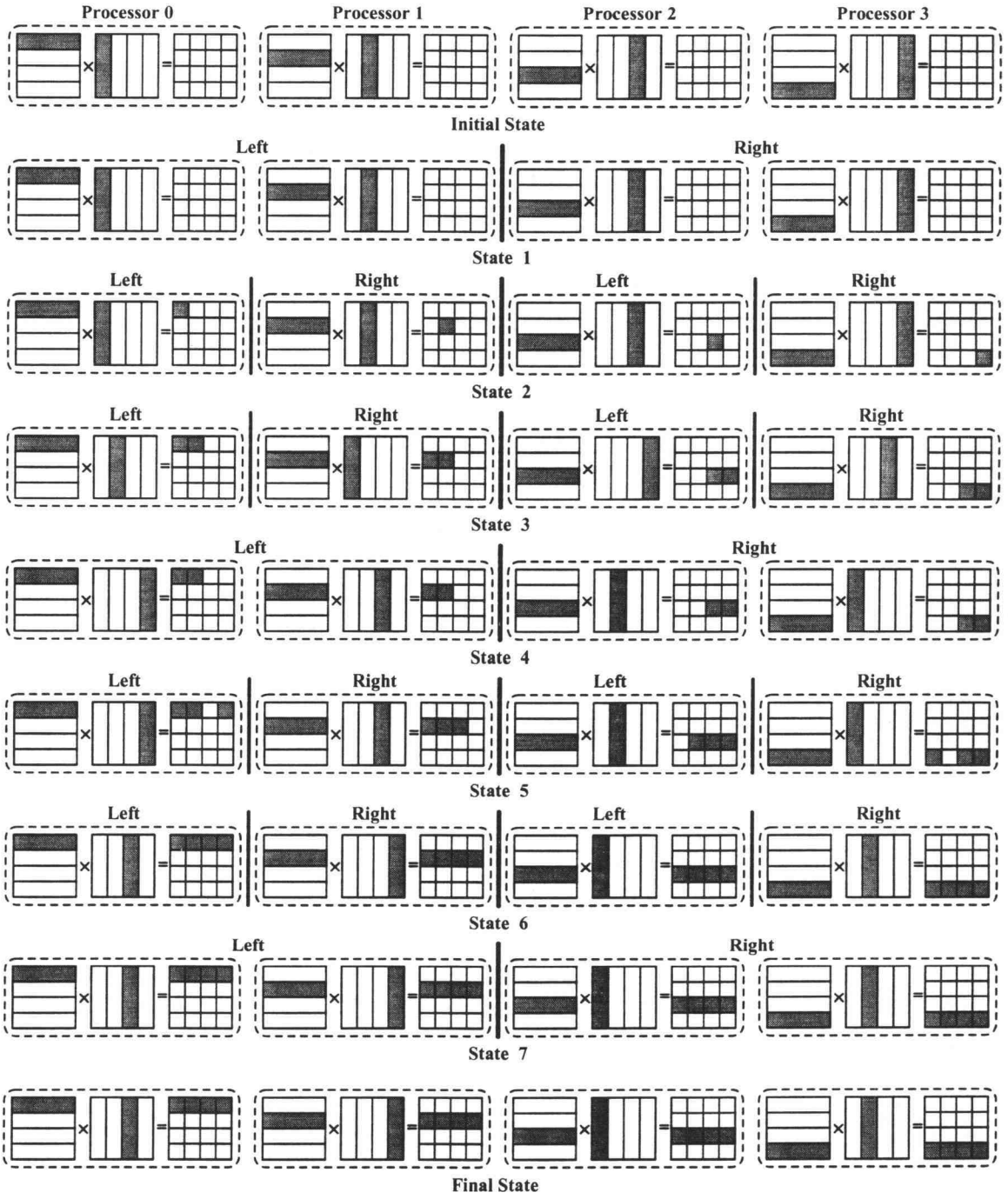

FIGURE 3.8. Snapshots of the data structures and processor partitions at the states labeled in the previous Figure. Shaded areas show the matrix blocks stored at a processor at the indicated state.

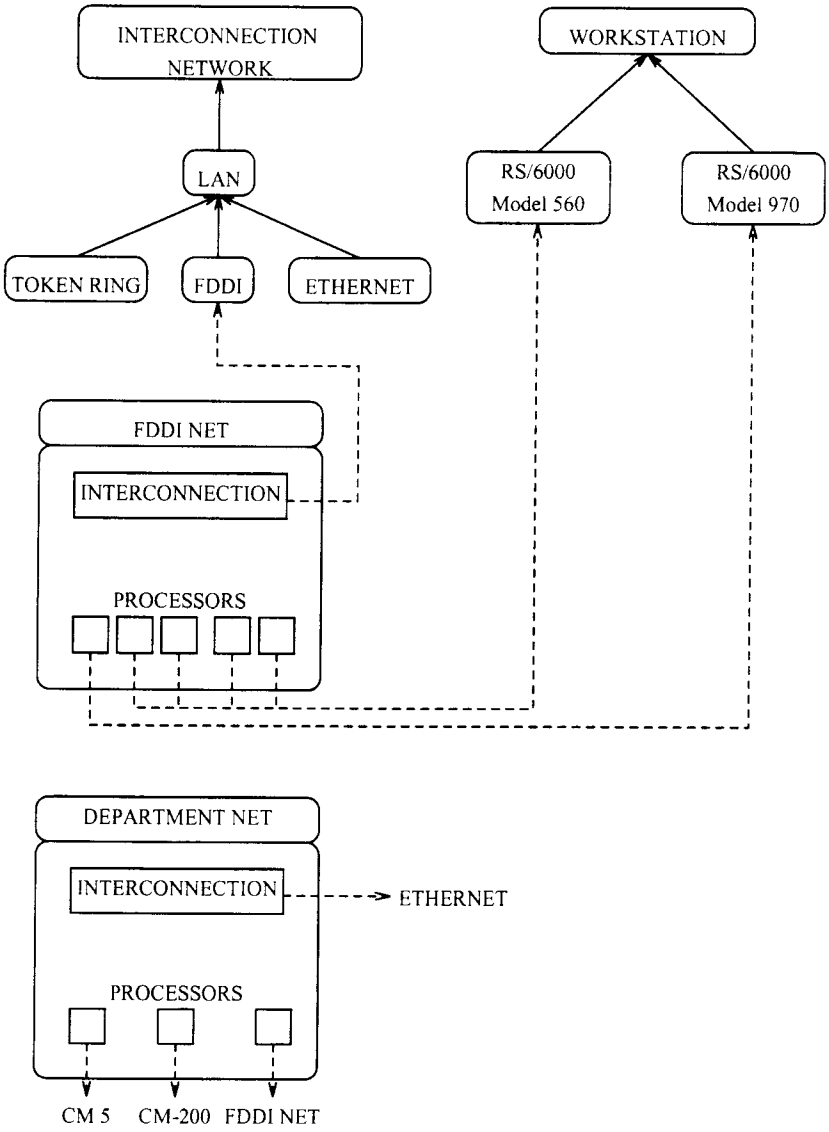

FIGURE 3.9. Frame representation of a typical processing environment.

We begin by introducing the terminology used for describing the model.

TERMINOLOGY:

$\vec{n}$	input size vector
p	number of processors
k	number of subproblems generated by each divide operation
$\vec{n}_s$	input size vector of the subproblems
$f()$	sequential time
$S()$	speedup
T_{par}	parallel time
T_{comp}	computation time
T_{comm}	communication time
C_{comm}	communication time for combine phase
D_{comm}	communication time for divide phase
C_{comp}	computation time for combine phase
D_{comp}	computation time for divide phase
C_{sync}	synchronization time for combine phase
D_{sync}	synchronization time for divide phase

The input size is a vector since some problems may have more than one input parameter. An example is the banded linear system solver, which has three input parameters: the number of unknowns, the bandwidth, and the tolerance. The input size vector of the subproblems can be formulated as a function of the original vector and the number of subproblems ($\vec{n}_s = g(\vec{n}, k)$). This function is very simple in most cases: the input size of the subproblems is obtained by simply dividing the input size of the original problem by k. For some applications, such as the banded-

matrix vector multiplication shown in Figure 3.4, a slightly more complex function is required.

We make several simplifying assumptions in forming the performance prediction model:

- The divide tree is *complete* and *balanced.*
- The processing environment is homogeneous.
- The base functions are sequential.
- There is no overlapping between computation and communication.
- The effect of cache on the speedup is negligible.

The first three assumptions have no bearing on most processing environments, including the three used for the case study in the next section. Nevertheless, we plan to improve the model in future to include arbitrary trees, heterogeneous processing environments, and non-sequential base functions.

The SPMD programs generated by our system currently do not support the overlapping of communication and computation. In our computational model, the opportunity for such overlap is limited, since it can be done only in the adjust functions. We do not consider this to be a liability as there is empirical and analytical evidence to suggest that communication-computation overlap has limited benefits [37, 38].

Can we ignore the impact of cache on the performance? Since we are using performance prediction to compare implementations, we are interested in relative performance rather than absolute performance. As long as cache effects do not change the relative performance of the implementations we are comparing, it is safe to ignore them. Our experience with the system, including the case study

presented in this paper, validates this assumption. However, cache effects could be significant for certain applications and architectures. Refining the performance prediction model to include memory effects is one of our future goals, especially since our system has access to the required information—such as the input size, the memory access pattern of the application, and the memory hierarchy of the architecture.

Predictor functions: Attached to each template is a function to compute its predicted performance. We will call this the *predictor* function. Parameters of the template are arguments of this function. Additionally, the predictor function has an extra argument denoting the type of the architecture. There are only three permissible values for this argument. These values and the associated architecture type are listed below:

1. S: Shared Memory machines.

2. X: Distributed memory platforms with an eXclusive access communication Medium.

3. N: Distributed memory platforms with a Non-exclusive access communication Medium.

When the *Performance Predictor* receives the instantiated template from the *Search Engine* (see Figure 3.3), it invokes the predictor function with the appropriate values for the arguments. The predictor function returns an expression, which encapsulates the predicted performance in a format independent of the specific details of the architecture. The Performance Predictor uses the machine parameters it received from the Search Engine to reduce this expression to a number representing the predicted *speedup*.

The predictor function essentially evaluates a small set of expressions, some of them defined recursively. At the top level, this function is the same for all templates and is defined by the following set of equations with the first entry in each equation showing the set of architecture classes to which it is applicable:

$$\{\text{S,X,N}\}\quad S(\vec{n},p) = f(\vec{n})/T_{par}(\vec{n},p)$$

$$\{\text{X,N}\}\quad T_{par}(\vec{n},p) = T_{comp}(\vec{n},p) + T_{comm}(\vec{n},p)$$

$$\{\text{N}\}\quad T_{comm}(\vec{n},p) = \begin{cases} 0 & (p=1) \\ T_{comm}(\vec{n}_s,p/k) + D_{comm}(\vec{n},p) + C_{comm}(\vec{n},p) & (p>1) \end{cases}$$

$$\{\text{X}\}\quad T_{comm}(\vec{n},p) = \begin{cases} 0 & (p=1) \\ k \times T_{comm}(\vec{n}_s,p/k) + D_{comm}(\vec{n},p) + C_{comm}(\vec{n},p) & (p>1) \end{cases}$$

$$\{\text{S}\}\quad T_{par}(\vec{n},p) = T_{comp}(\vec{n},p) + T_{sync}(\vec{n},p)$$

$$\{\text{S}\}\quad T_{sync}(\vec{n},p) = \begin{cases} 0 & (p=1) \\ k \times T_{sync}(\vec{n}_s,p/k) + D_{sync}(\vec{n},p) + C_{sync}(\vec{n},p) & (p>1) \end{cases}$$

$$\{\text{S,X,N}\}\quad T_{comp}(\vec{n},p) = \begin{cases} f(\vec{n}) & (p=1) \\ T_{comp}(\vec{n}_s,p/k) + D_{comp}(\vec{n},p) + C_{comp}(\vec{n},p) & (p>1) \end{cases}$$

The equations above merely reflect the structure of the divide-and-conquer template and hence remain the same for all templates. The predictor function of each template will have additional expressions to compute the application-specific details. Next we discuss these details and the methods used for computing them.

- $f(\vec{n})$. We need the number of scalar floating point operations, the number of vectorizable loops, and the number of vector operations in each such loop of

the sequential base function to compute $f(\vec{n})$. The predictor function of each template computes these numbers using the input size vector $\vec{n}$.

We assume that the vectorizable loops in the base function can be identified without prior knowledge of the target platform. In reality, a loop that vectorizes on one vector machine may not vectorize on another machine, primarily due to variations in the compiler technology. Since the templates would be developed by experts rather than novice users, we assume the base functions are coded in such a way that most compilers can vectorize them.

- $D_{comp}(\vec{n}, p)$ and $C_{comp}(\vec{n}, p)$. Just as in the previous case, the predictor function computes the scalar and vector operation counts using the input size vector and the number of processors. These numbers correspond to computations in the adjust functions of the template.

- $D_{comm}(\vec{n}, p)$ and $C_{comm}(\vec{n}, p)$. We make use of the structure of divide-and-conquer to formulate these expressions in an architecture-independent manner. The predictor function simply returns a list of communication operations in the adjust functions of the template. These operations are well-defined system primitives. Examples include *correspondence communication* and *mirror image communication.* In addition to the identifier of an operation, the list entries will also include the values of the parameters of this operation. An example is shown in Figure 3.10.

ID: CORR (Correspondence communication)
Parameters:
DIRECTION: LR (Left sends to the Right)
DATA SIZE: 40 Bytes
PROCESSORS: 8 (Four on the Left and four on the Right)

FIGURE 3.10. An example entry in the list of communication operations.

- $D_{sync}(\vec{n}, p)$ and $C_{sync}(\vec{n}, p)$. The predictor function simply returns the number of synchronizations required in the adjust functions in a shared-memory environment as the values of these expressions.

Notice that the predictor function is not computing the execution times directly. This is accomplished by the *Performance Predictor* using machine-specific details. We will show how this is done for the communication time component. Since communication primitives are only few in number, the cost function for each such primitive is stored explicitly in the machine database. Since the Performance Predictor knows the specific target machine of the template, it invokes the appropriate cost function of this machine for each entry in the list it receives from the predictor function. Computation and synchronization times are computed similarly, by combining the expressions returned by the predictor function with machine parameters.

The two-step computation of predicted performance described above has the advantage of decoupling the templates from the architectural details, while maintaining great flexibility in analytical performance prediction.

3.5. Case Study: Conjugate Gradient

We present an example in which the scheme described in the previous sections is used to develop efficient parallel implementations of the conjugate gradient (CG) method for three diverse architectures.

3.5.1. Mathematical Description of the CG Method

The conjugate gradient method is an iterative scheme for solving linear systems of equations. Given a symmetric, positive definite, coefficient matrix A, and a vector b, it computes the solution vector of the linear system $Ax = b$ using the algorithm shown in Figure 3.11 [39].

$i = 0; g_i = h_i = b - Ax_i;$
while (not converged) do:
$\lambda_i = {g_i}^T h_i / ({h_i}^T A h_i)$
$x_{i+1} = x_i + \lambda_i h_i$
$g_{i+1} = b - Ax_{i+1}$
$\gamma_i = (g_{i+1} - g_i)^T g_{i+1} / ({g_i}^T g_i)$
$h_{i+1} = g_{i+1} + \gamma_i h_i$

FIGURE 3.11. Conjugate gradient algorithm.

3.5.2. An Algorithmic Template for CG

Each CG iteration involves a matrix vector multiplication and a few dot products and SAXPYs. Each one of these operations can be expressed using a divide-and-conquer template. Thus the CG method is represented in our scheme as a meta-template with several constituent divide-and-conquer templates.

The meta-template is parameterized using the following three parameters:

1. Processors used for matrix-vector multiplication (P1). The matrix vector multiplication is the most compute-intense task in the CG iteration. Hence it will be beneficial to use all the available processors for this operation. Thus, the size of the target platform essentially determines this parameter.

2. Processors used for the rest of the operations (P2). The poor granularity of dot product can affect the overall performance of the CG implementation. This

parameter would let us improve the granularity by computing dot product on a subset of the available processors. The system uses performance prediction to decide the optimum granularity depending on the machine characteristics and problem size.

The computational complexity of the CG iteration is concentrated in the matrix-vector multiplication. This is an $O(n^2)$ operation, while the rest of the computation has only linear time complexity. By tying together the granularities of all linear-time operations, we reduce the number of parameters and thereby limit the size of the search space. In contrast, if we allow the granularity of each individual operation to change independently, the search space will have a combinatorial explosion. But this will not necessarily lead to a better solution, since the cost of redistributing the data (in the distributed-memory machines) or synchronizations (in the shared-memory machines) will eventually force the search engine to choose an implementation with the same granularity for these operations.

This is an example of the pruning of the search space using application-specific knowledge. As mentioned in section 3, an alternate method is to start with a full set of parameters and then use branch-and-bound algorithms to eliminate unproductive branches of the search tree.

3. Decomposition of the coefficient matrix (MAT). The distribution of the coefficient matrix among the processors is an important parameter, since the adjust functions, the divide function, and the combine function will be determined by this distribution. We consider three different distributions:

   (a) Row-contiguous.

   (b) Column-contiguous.

(c) Block-contiguous.

In Figure 3.12, we present a parameterized meta-template for a single iteration of CG using pseudo-code.

The CG-Template in Figure 3.12 is called a *meta-template* since it is built by the composition of a number of *simple* divide-and-conquer templates. This example illustrates our *layered* approach to building adaptable parallel programs using divide-and-conquer.

The base templates used for building meta-templates fall into two categories on the basis of their functionality:

1. Basic linear algebra templates. Examples include matrix multiplication, dot product, and matrix vector multiplication.

2. Data redistribution templates. Each linear algebra template has a set of pre-conditions and post-conditions, which are specified in terms of the distribution of the input and output data structures respectively. When two templates are concatenated to form a meta-template, it might be necessary to redistribute the output data from the first to meet the pre-conditions on the second. Data redistribution templates are used for this purpose. Notice that these will be required only for distributed memory machines.

Pre-conditions:
Matrix A distributed as specified by the parameter **MAT**
Vector h distributed on **P1** nodes
Vector g distributed on **P2** nodes
Vector x distributed on **P2** nodes
Vector b distributed on **P2** nodes

CG-Template(P1,P2,MAT)
switch (**MAT**)
case **Row-contiguous**:
Invoke DC-Template for row-oriented matrix vector multiply
with **P1** as the number of processors to use.
Invoke DC-Template to distribute the product vector from
P1 nodes to **P2** nodes.
case **Column-contiguous**:
Invoke DC-Template for col-oriented matrix vector multiply
with **P1** as the number of processors to use.
Invoke DC-Template to distribute the product vector from
one node to **P2** nodes.
case **Block-contiguous**:
Invoke DC-Template for block-oriented matrix vector multiply
with **P1** as the number of processors to use.
Invoke DC-Template to distribute the product vector from
one node to **P2** nodes.
End *switch* (**MAT**).

Invoke DC-Template to distribute the vector h from **P1** nodes to **P2** nodes.

Invoke a series of DC-Templates for dot product and SAXPY
with **P2** as the number of processors to use.

Invoke DC-Template to redistribute the vector h from **P2** nodes
to **P1** nodes.
End **CG-Template**.

FIGURE 3.12. A parameterized meta-template for a single iteration of CG.

All constituent base templates of the CG-template utilize either the Left-Right divide (LR) or the LEFT-RIGHT-TOP-BOTTOM divide (LRTB), the two standard divide functions. The LR divide splits the processor pool into two equal partitions, a LEFT partition and a RIGHT partition. It imposes a linear ordering on the processors. Figure 3.13 shows the LR divide of eight processors. The LRTB

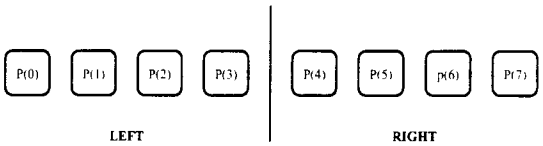

FIGURE 3.13. LEFT-RIGHT division of 8 processors.

divide splits the processors into four partitions, LEFT-TOP, LEFT-BOTTOM, RIGHT-TOP, and RIGHT-BOTTOM. The processors are arranged logically as a square 2D mesh as shown in the Figure 3.14, where the LRTB divide is applied on a pool of 16 processors.

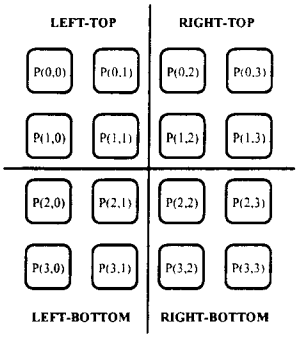

FIGURE 3.14. LEFT-RIGHT-TOP-BOTTOM division of 16 processors.

3.5.2.1. Linear Algebra Templates

Below we describe the linear algebra templates used for building the CG template:

- **Row-oriented Matrix Vector Multiplication** : $Ab = c$. All data structures are distributed among the processors with row-contiguous distribution used for the matrix.

  Divide function: LR.

  Pre-adjust function: None.

  Base function: Sequential Matrix Vector Multiplication.

  Post-adjust function: (1) Swap values of b between partitions. (2) Recursively apply Template to both partitions.

- **Column-oriented Matrix Vector Multiplication** : $Ab = c$. The input data structures are distributed among the processors with column-contiguous distribution used for the matrix. On completion, vector c will be accumulated on a single processor.

  Divide function: LR.

  Pre-adjust function: None.

  Base function: Sequential Matrix Vector Multiplication.

  Post-adjust function: The first processor on the LEFT gets the vector c from its counterpart on the RIGHT and adds it to its own c.

- **Block-oriented Matrix Vector Multiplication** : $Ab = c$. The matrix is distributed block-wise among the processors arranged in a 2-D mesh. The input vector b is distributed among the processors in column-major order. The output vector c is accumulated at a single processor. In the pre-adjust phase,

subvectors are assembled at each node. In the post-adjust phase, the partial results are combined and accumulated.

Divide function: LRTB.

Pre-adjust function: Get the chunk of b from the *vertical* counterpart and store it at the appropriate location within the subvector being assembled.

Base function: Sequential Matrix Vector Multiplication.

Post-adjust function: (1) The FIRST processor in the LEFT-TOP and the FIRST processor in the LEFT-BOTTOM get c from their counterparts on the RIGHT and add it to their own c. (2) The FIRST processor in the LEFT-TOP gets the c from its counterpart on the BOTTOM and concatenates it to its own c.

- **Dot Product**: $d = ab^T$. The input vectors are distributed among the processors and the output is replicated at each processor.

  Divide function: LR.

  Pre-adjust function: None.

  Base function: Sequential dot product.

  Post-adjust function: Each processor adds its counterpart's partial result to its own.

- **SAXPY**: $b = \alpha x + y$.

  Divide function: LR.

  Pre-adjust function: None.

  Base function: Sequential saxpy.

  Post-adjust function: None.

3.5.2.2. Data Redistribution Templates

We have used two data redistribution templates to form the CG-template.

- **Vector Distribution.** A vector stored at a single processor is distributed among all processors using this template.

  Divide function: LR.

  Pre-adjust function: The first processor on the LEFT sends the latter half of its vector to the first processor on the RIGHT.

  Base function: None.

  Post-adjust function: None.

- **Vector Concatenation.** A vector distributed among all processors is concatenated and stored at a single processor using this template.

  Divide function: LR.

  Pre-adjust function: None.

  Base function: None.

  Post-adjust function: The first processor on the LEFT gets the subvector from its counterpart on the RIGHT and concatenates this subvector to its own.

3.5.3. Generation of Efficient Programs on Diverse Platforms

The conjugate gradient template is adapted to a specified target platform by tuning the values of the parameters described earlier. The *Search Engine* module of the system will invoke the *Performance Predictor* several times to determine the set of parameters that maximizes the speedup. For a machine with p processors, there are only $3(\log p + 1)$ leaves in the search tree, making exhaustive search possible.

In this case study, we have considered three vastly different parallel processing environments as target platforms: an eight-processor Silicon Graphics Power Challenge shared memory machine, a 32-processor CM-5, and an FDDI network of four IBM RS/6000 model 560 workstations.

3.5.3.1. Adapting the Template to a Shared Memory Machine

Figures 3.15, 3.16, and 3.17 show the predicted and observed speedups for an input size of 1024 for row, column, and block distribution of the matrix respectively. The predictions tend to be more optimistic than the actual performance, but in terms of the relative performance, the predicted values match with observations.

We will show how the Search Engine can make the correct decision using the performance prediction by closely examining the data for the 4-processor machine. Figure 3.18 shows the predicted and observed performance for the nine possible combinations of the parameter values. These combinations are formed by the cartesian product of the sets { 1, 2, 4} and {ROW, COLUMN, and BLOCK}. The first set denotes the permissible values of the parameter $P2$, the number of processors to be used for the dot product computations. The elements in the second set represent the three matrix decomposition options.

The best performance is predicted when $P2 = 4$ with row decomposition of the coefficient matrix. The actual implementations showed maximum speedup for the same values of these parameters.

On an 8-processor machine, predictions showed row decomposition of the matrix along with $P2 = 4$ as the best choice of parameter values. These values were also validated by the observations.

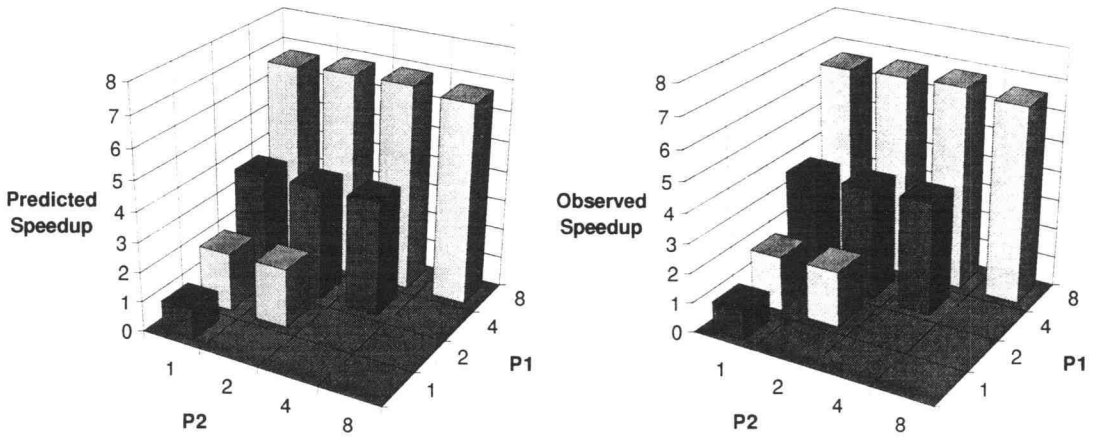

FIGURE 3.15. Predicted and observed performance of CG on SGI using ROW decomposition of the matrix. P1 is the number of processors used for matrix vector multiplication and P2 is the number of processors used for the rest of the computation.

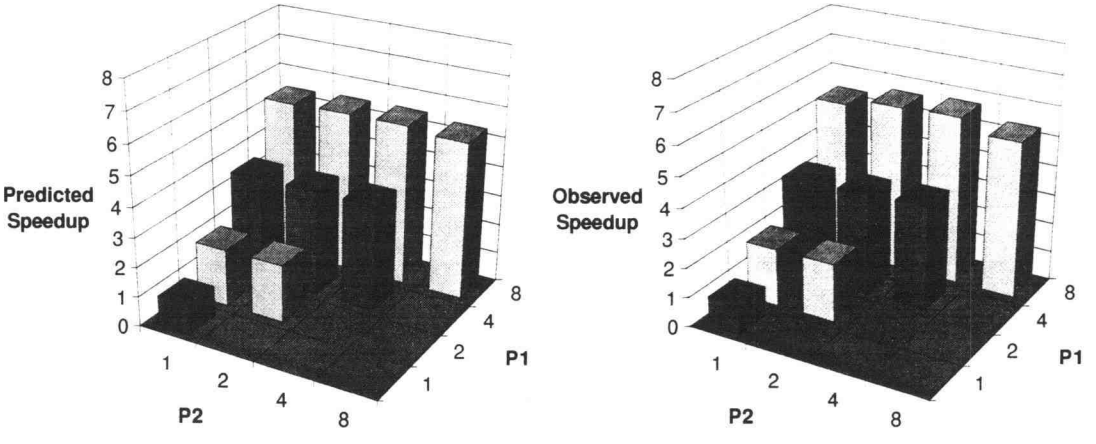

FIGURE 3.16. Predicted and observed performance of CG on SGI using COLUMN decomposition of the matrix.

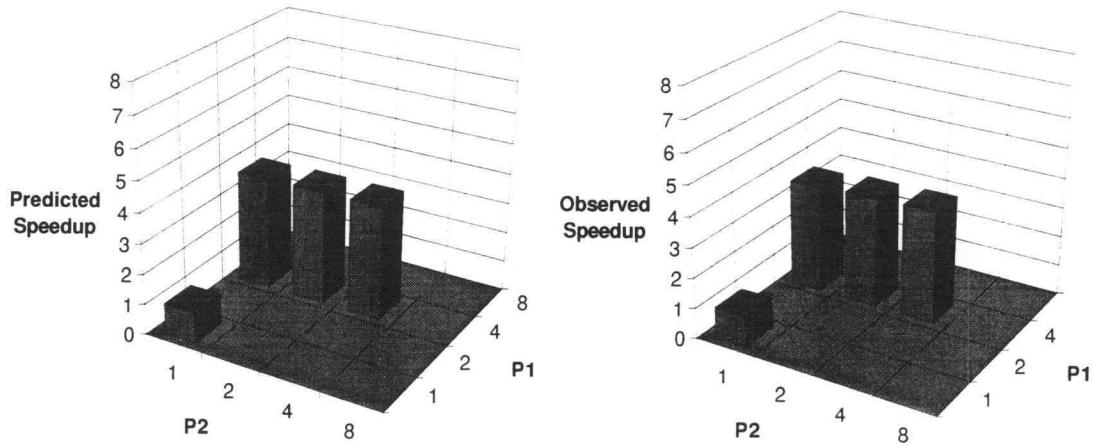

FIGURE 3.17. Predicted and observed performance of CG on SGI using BLOCK decomposition of the matrix.

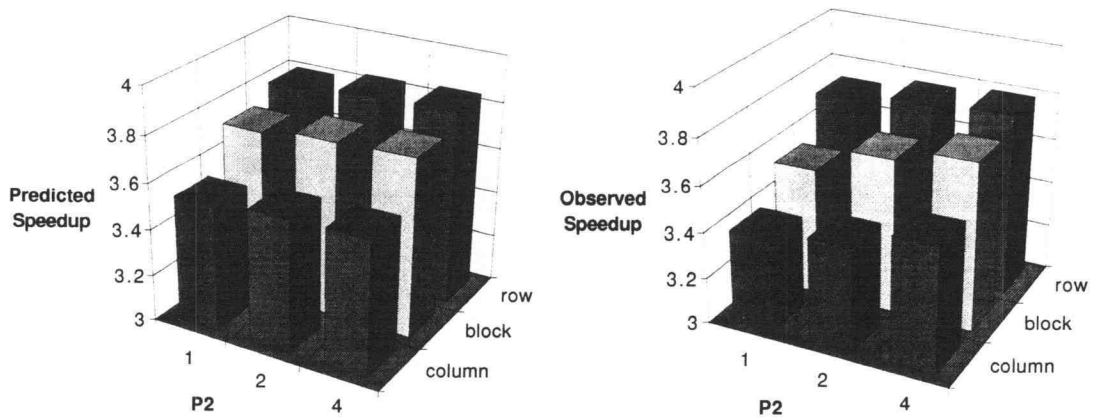

FIGURE 3.18. Predicted and observed performance of CG on a 4-processor SGI for row, column, and block decomposition of the matrix. ($P1 = 4$.)

In general, the Search Engine can make the following decisions based on the performance prediction:

- The best decomposition scheme for the coefficient matrix is *row-contiguous.*
- It is advantageous to use all processors for matrix vector multiplication.
- The rest of the computation is at best performed using four or fewer processors.

An explanation of these decisions is straightforward. The implementation that minimizes the synchronization points performs best on a shared memory machine. Since row-wise matrix vector multiplication requires no synchronizations at all, this scheme beats the other options easily. It is advantageous to utilize all the available processors for matrix vector multiplication, since this is a rather compute-intensive task. The granularity of the dot product computation that maximizes the performance is a function of the vector size as well as the machine parameters. For a vector size of 1024, four processors minimizes the execution time.

3.5.3.2. Adapting the Template to a Workstation Network

Figure 3.19 shows the predicted and observed performance using row decomposition of the matrix for varying cluster sizes and dot product granularities. The problem size is kept the same as in the shared memory example—1024 unknowns. Figure 3.20 shows the same information when the matrix is decomposed column-wise. Performance of the CG scheme using block distribution of the coefficient matrix is shown in Figure 3.21. The predictions enable the system to arrive at the best implementation on the workstation network: column decomposition of the coefficient matrix along with single processor execution of the dot product.

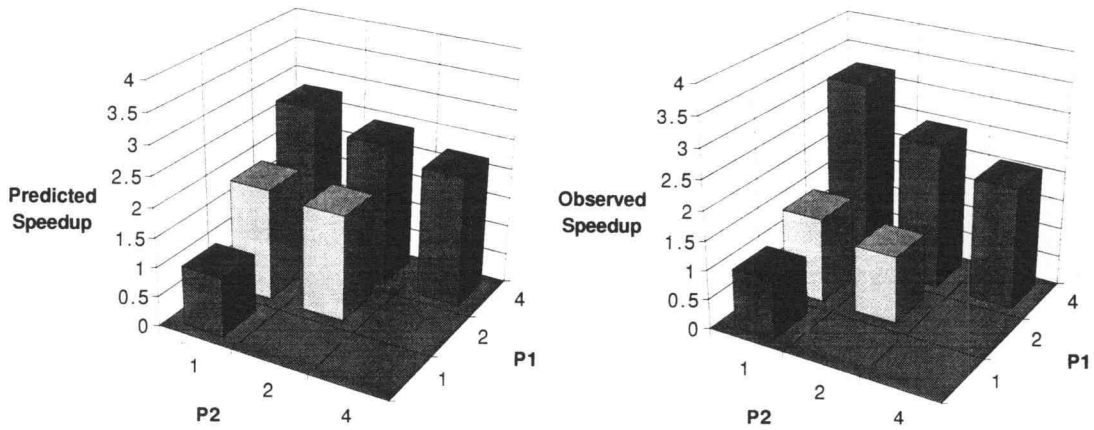

FIGURE 3.19. Predicted and observed performance of CG on workstation network using ROW decomposition of the matrix. P1 is the number of processors used for matrix vector multiplication and P2 is the number of processors used for the rest of the computation.

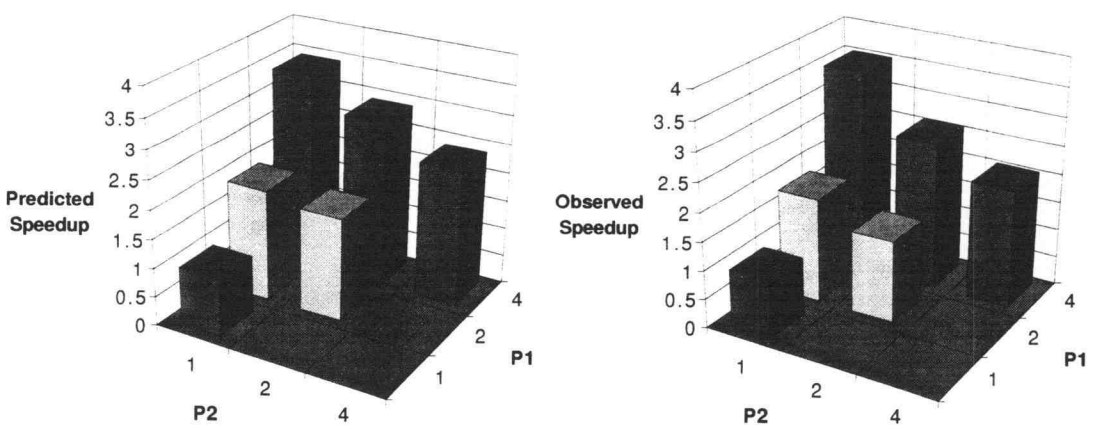

FIGURE 3.20. Predicted and observed performance of CG on workstation network using COLUMN decomposition of the matrix.

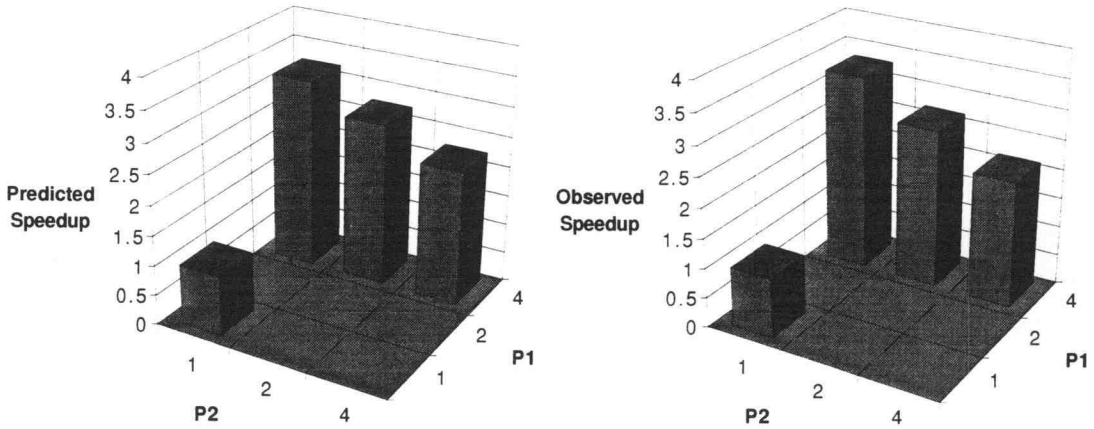

FIGURE 3.21. Predicted and observed performance of CG on workstation network using BLOCK decomposition of the matrix.

In the workstation environment, the execution time is minimized by the algorithm that minimizes the total number of messages generated, since the shared nature of the communication medium forces all messages to be serialized. This explains the selection of column decomposition for the workstation network. The extremely coarse nature of the network discourages parallel execution of the dot product computation.

3.5.3.3. Adapting the Template to a Multicomputer

Figures 3.22, 3.23, and 3.24 show the predicted and observed speedups on a CM-5 using row, column, and block decomposition respectively. To see how predictions help in selecting the best values of the parameters, we will look closely at the data for a 16-processor machine and a 32-processor machine.

Figure 3.25 shows the predicted and observed performance for five different values of P2 and three different matrix decomposition options on a 16-processor

machine. Both predictions and observations agree on 16 as the best value of P2 and *block-contiguous* as the best matrix decomposition option.

The block decomposition of the matrix leads to the minimum number of communication steps in the algorithm and consequently to the best performance. The fine granularity of the machine justifies the spreading of the entire computation among all the available processors.

On a 32-processor machine, both predictions and observations show column decomposition outperforming row decomposition, with 32 as the optimal value of P2.

In general, we see that the analytical performance prediction helps the Search Engine in choosing the appropriate values for the parameters, leading to an implementation that maximizes performance.

3.5.4. Comparison with Other Parallel Programming Systems

In the previous section, we gave performance figures for a parallel CG solver on diverse architectures programmed based on our approach. How does our system compare with other commonly used parallel programming tools in terms of performance? To answer this question, we implemented the conjugate gradient method using CM Fortran on a CM-5 and Power Fortran on an SGI. Our best implementation on the 32 node CM-5 outperformed the CM Fortran program by a factor of 1.5. On the SGI multiprocessor, our performance was 1.7 times better than a Fortran program optimized using Power Fortran directives.

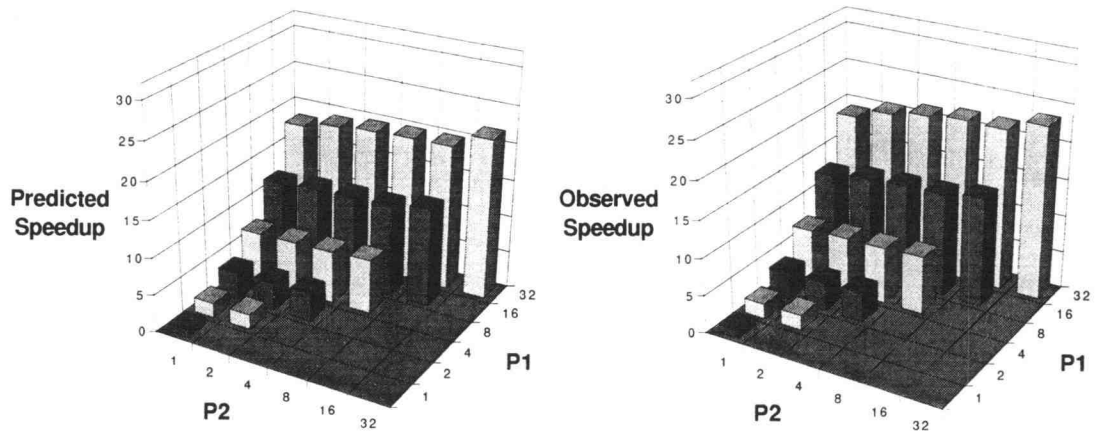

FIGURE 3.22. Predicted and observed performance of CG on CM-5 using ROW decomposition of the matrix. P1 is the number of processors used for matrix vector multiplication and P2 is the number of processors used for the rest of the computation.

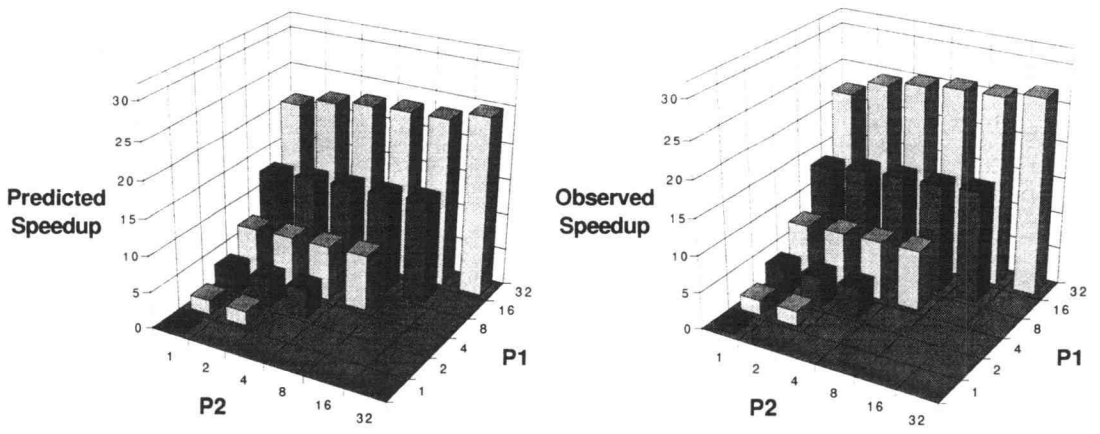

FIGURE 3.23. Predicted and observed performance of CG on CM-5 using COLUMN decomposition of the matrix.

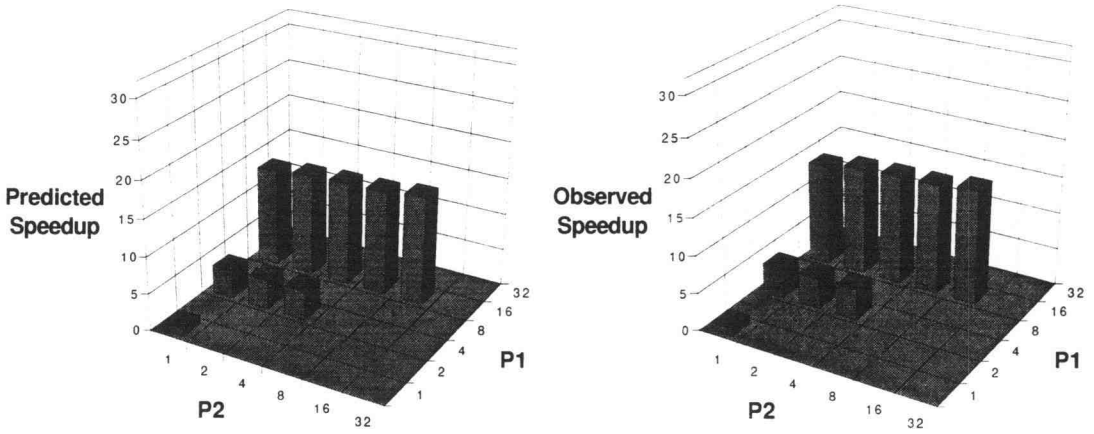

FIGURE 3.24. Predicted and observed performance of CG on CM-5 using BLOCK decomposition of the matrix.

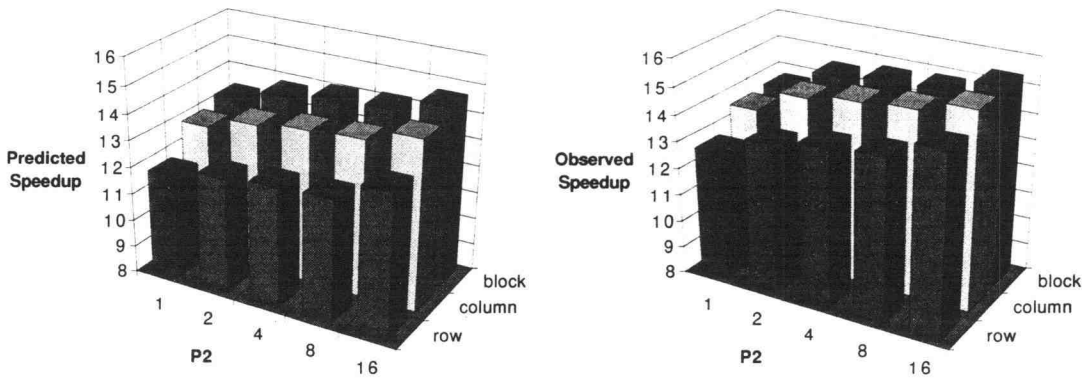

FIGURE 3.25. Predicted and observed performance of CG on a 16-processor CM-5 for row, column, and block decomposition of the matrix. ($P1 = 16$.)

3.5.5. Conclusions from the Case Study

Three key issues in parallel processing are performance, portability, and programmability. We believe that our method addresses all three of them, at the expense of generality. The loss in generality is not a serious drawback if the expressibility of the system is sufficient to cover most problems of practical interest. By using the conjugate gradient method as an example, we have shown that an algorithm of significant practical interest can be represented using our system.

As detailed in the previous section, the case study corroborates our thesis that performance prediction along with the divide-and-conquer paradigm can provide architecture adaptability.

The case study shows that the algorithms generated by the system can have efficient implementations on diverse processing environments. There are two reasons for this. First, the system is able to search the parameter space exhaustively, since this space is relatively small. The other reason for good efficiency is the low overhead of our implementation of the DC templates [5].

We have seen similar results from another case study involving a finite element ocean circulation model [9].

3.6. Code Generation

Hatcher and Quinn describe a compiler for a data-parallel language targeted towards MIMD computers in [2]. Their compiler generates SPMD programs on distributed-memory machines and shared-memory machines from data-parallel specifications. The task of our **code generator** is similar, albeit much simpler. In this

section, we outline the implementation of the templates, and the mechanisms for handling communication calls and data distribution primitives.

3.6.1. Template Implementation

We use a set of higher-order functions to implement the templates. The base templates fall into four categories based on the presence or absence of the adjust functions. Templates in each category are implemented using a function associated with that category. Below we describe these categories and the associated functions:

1. Templates with pre-adjust and post-adjust functions. The associated function is called **PDC**. An example of this type of template is the block-oriented matrix multiplication. The code in Figure 3.26 shows the implementation of the PDC function in C.

   The base function, the divide and combine functions, and the adjust functions are passed as arguments to the higher-order function **PDC**. Additionally, the number of processors and the dimension of the processor grid, along with a pointer to the application data and a pointer to the system data are also passed as arguments. The dimension of the processor grid refers to the logical 1-D or 2-D mesh embedded in the topology of the machine. Divide and combine functions assume the existence of such an embedding. The system data structures remain the same for all templates. They simulate the traversal up and down the divide-and-conquer tree. The application data will change for each template, since they are specific to the problem being solved.

   The routine first computes the depth of the divide-and-conquer tree. Here we assume the default base predicate; the recursion terminates when there is only a single processor in each partition.

```
void PDC( basefun, dfun, cfun, prafun, posfun, p, dim,
                Tptr, DCptr )
void (*basefun) (); /* base function */
void (*dfun)    (); /* divide function */
void (*cfun)    (); /* combine function */
void (*prafun)  (); /* pre-adjust function */
void (*posfun)  (); /* post-adjust function */
int  p;             /* number of processors */
int  dim;           /* dimension of the processor grid */
struct TEMPLATE_DATA * Tptr;  /* pointer to app data */
struct DC_STATE * DCptr;      /* pointer to system data */
{
 int i, depth,tmp;
 tmp = 1;
 depth = 0;
 /* compute the depth of the DC-tree */
 while (tmp < p) {
  depth++;
  tmp <<= dim;
 }
 /* divide them with pre-adjusting */
 for (i=0; i<depth; i++) {
    (*dfun)(DCptr);
    (*prafun)(DCptr,Tptr);
 }
 /* activate the base function now  */
 (*basefun)(DCptr,Tptr);
 /* combine them  (with post-adjusting)  */
 for (i=0; i<depth; i++) {
    (*posfun)(DCptr,Tptr);
    (*cfun)(DCptr);
 }
 return ;
}
```

FIGURE 3.26. C function PDC.

Since depth of the divide tree is known *a priori*, we replace the recursion by two iterative loops with a call to the base function placed between them. The first loop simulates going down the divide tree from the root to the leaves. At the leaves, we invoke the base function. The second loop simulates going up the tree from the leaves to the root.

2. Templates with only pre-adjust functions. We call the associated function **prePDC**. The vector distribution template presented earlier is an example of this category. This function is derived from **PDC** by replacing the second iteration by a single function call, which has the effect of transforming the state from the leaves to the root in a single operation.

3. Templates with only post-adjust functions. Similar to the previous case, we replace the first iteration by a function call to derive **postPDC** from **PDC** to represent this category. Most templates presented earlier in this paper are examples of this type.

4. Templates with no adjust functions. We call the associated function **purePDC**. The SAXPY operation used earlier in the conjugate gradient template is an example of this category.

3.6.2. Communication

The inter-processor communication is limited to the adjust functions. These communications appear as calls to a handler function in the pre-adjust and post-adjust functions. The handler functions are responsible for generating machine-specific communication calls. The divide-and-conquer paradigm has the advantage

of requiring regular communication patterns. The handler functions exploit this regularity to generate machine-specific, efficient implementations.

Each divide function has an associated set of patterns. We present below some of the most frequently used communication patterns for the 1-D left-right divide operation:

- Correspondence Communication. This pattern is shown in Figure 3.27 below. Each processor communicates with the corresponding processor on the other partition.

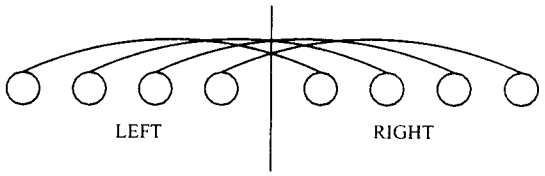

FIGURE 3.27. Correspondence communication with LEFT-RIGHT division.

- Mirror Image Communication. As shown in Figure 3.28, each communication link is symmetrical with respect to the centerline.

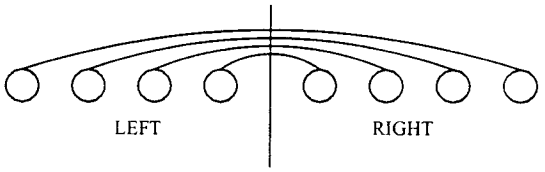

FIGURE 3.28. Mirror image communication with LEFT-RIGHT division.

- Between the Last processor on the LEFT partition and the First processor on the RIGHT partition. This pattern results in neighbor communication, as shown in Figure 3.29.

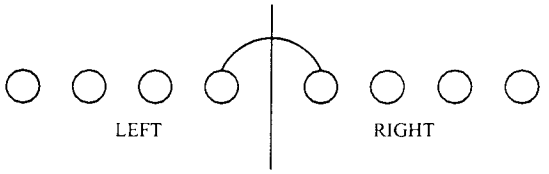

FIGURE 3.29. Last to First communication with LEFT-RIGHT division.

- Between the First processor on the LEFT partition and the First processor on the RIGHT partition. This pattern is shown in Figure 3.30.

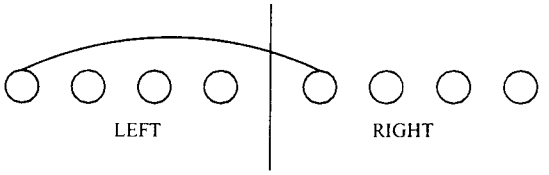

FIGURE 3.30. First to First communication with LEFT-RIGHT division.

Each one of the above patterns have three different variations based on the message direction: *Left to Right*, *Right to Left*, or *Duplex*. The information passed to the handler function includes a pattern identifier, message direction, address at the source node, address at the target node, and the size of data being transferred.

On shared-memory machines, the shared memory can be used for information exchange. The source node writes to the appropriate locations in the shared memory, and the target node reads them. But unlike in message-passing architectures, processors should synchronize to ensure that a *read* does not happen before a *write*. It is the responsibility of the handler function to insert the necessary synchronization calls.

3.6.3. Data Distribution Primitives

A meta-template may include several data distribution primitives to ensure compatibility between the constituent base templates. We have used two such primitives in the CG-Template: vector concatenation and vector distribution. These primitives themselves are implemented as divide-and-conquer templates. But unlike other base templates, these are required only on distributed memory machines. On platforms with *shared memory*, there is clearly no need for explicit data redistribution and the Code Generator suppresses these calls.

3.7. User Interface

Our goal is to isolate the computational scientist from the details of parallel hardware and algorithms. This is achieved by having a very clear delineation of roles for the participants. We briefly describe these roles below:

1. Environment builder: The role of the environment builder is to develop the enabling technology. This includes the user interface, search engine, and the performance prediction tools.

2. Environment maintainer: The primary role of the environment maintainer is to extend the problem solving capabilities of the system by generating appropriate algorithmic templates and by providing the necessary parameters to drive the performance predictor.

3. Computational scientist: At the highest level, the users interact with the system by specifying the problem to be solved using a high-level notation. Thus, the computational scientist is able to concentrate on the science, without worrying about the details of the underlying computing environment.

The high-level notation mentioned above serves as the user interface. There are three important criteria for the selection of this notation:

1. The computational scientist should be familiar and comfortable with the notation.

2. It should be possible to represent computational science problems easily using the notation.

3. It should be possible to extract templates easily from "programs" written in the notation.

A good candidate for the user interface is the notation used by MATLAB, a well-known software package for numerical computation, data analysis, and graphics [4]. Complex numerical applications can be coded using the MATLAB notation. A front end will scan and parse the MATLAB programs, and generate an intermediate representation consisting of calls to high-level functions. For each function call, the *Template Selector* shown in Figure 3.3 will then search the database for matching templates.

3.8. Related Work

Our performance prediction model is based on the work done by Clement and Quinn in predicting the performance of scalable data-parallel programs on multicomputers [36]. Parashar *et al.* have proposed the use of performance prediction to improve the performance of parallel programs [40]. As part of a HPF/Fortran 90D application development environment, their performance prediction framework helps users in selecting appropriate compiler directives.

The algebraic model of divide-and-conquer, introduced by Mou and Hudak in [23], has influenced the design of our templates. Chandy's group at Caltech [41] and Dongarra's group at Tennessee [42] are also investigating the use of "templates" for high performance scientific computing. Our method differs from these and other work on templates by introducing a novel approach to architecture adaptability, combining parameterized templates with analytical performance prediction.

A vast amount of research in parallel programming has been motivated by the desire to make parallel programming easier and portable. Here we attempt to categorize these efforts and comment on their impact in the real world.

Architecture-independent programming languages and systems: A high-level parallel programming language can provide limited architecture independence and programmability if efficient compilers are available on several machines. Dataparallel C [2] and Fortran 90 [43] are two examples. Chameleon [44] is a shared memory library designed for architecture independence. Several message passing libraries—most notably PVM [45], MPI [46], and p4 [47]—are in existence to facilitate portable parallel programming on distributed-memory machines. Crowl's Matroshka system presents the framework of a parallel programming language which has the ability to adapt to different architectures [48]. A Matroshka-based program exposes all the available parallelism in an application. Using annotations, a subset of this parallelism is selected for execution on a specific machine.

Selection of the appropriate algorithm is still up to the user when a general-purpose, procedural language is used for problem solving in parallel and distributed environments. This implies that effective portability is not achieved. For sequential machines, since there is only one machine model, this is not a problem. For distributed computing, this difficulty impedes the development of easily portable problem solving environments. The message passing libraries (such as PVM, MPI, p4, and

Illinois Fast Messages [49]) merely become accessories to our system since our goal is the automation of algorithm design and implementation using the available programming tools.

Another approach to designing machine-independent parallel programming relies on implicit parallelism. The parallel programming languages based on the functional paradigm fall into this catergory. A number of such languages have been proposed, including EPL [50], Crystal [51], ParAlfl [52], SISAL [53], Id [54], and a parallel dialect of Haskell [55]. A functional language can provide a higher level of abstraction to the programmer, compared to explicitly parallel procedural languages. But the objective of our work is to provide to users a much higher level of abstraction. We are striving for a parallel processing environment where the users are able to concentrate on the problems being solved, rather than the selection of the algorithms and their implementation in some *programming* paradigm. The choice of the programming paradigm, whether it is functional or procedural, is of little help to the computational scientist in designing the algorithm.

The object-oriented programming paradigm has also made an impact on machine-independent parallel programming. Several object-oriented parallel programming languages, systems, and models are in existence, including Mentat [56], pC++ [57], Concurrent Smalltalk [58], Illinois Concert System [59], Charm++ [60], Compositional C++ [61], and Actors [62]. Object technology makes it easy to separate the interface from the implementation. By hiding the architecture-specific parts in the implementation, an object-oriented parallel program will give the users a machine-independent interface. Several parallel class libraries (e.g. [63], [64], and [65]) and object-oriented frameworks (e.g. [66], [93], and [68]) have been built to exploit the software-engineering advantages of object technology. A key difference between our methodology and the object-oriented approaches mentioned above is

that we are directly attacking the difficult problem of generating optimized implementations of object-oriented programs.

Algorithm architecture mapping: Representing the algorithms and architectures as task graphs, graph embedding can be used to generate parallel programs that adapt to machine topologies. The Oregami project [69] is an example of tool development using this approach.

In practice, graphical representations of parallel programs are complex and may contain several communication patterns intermingled. Further, with the advance of wormhole routing, communication overheads are dominated by message startup times rather than the distance between the communicating processors or link congestion. In short, complex task graphs make this approach impractical, and the change in technology renders it irrelevant.

Multiprocessor scheduling: Another attempt at solving the problem resulted from looking at parallel processing as precedence-constrained multiprocessor scheduling with interprocessor communication delays. The tools developed by El-Rewini and Lewis for "scheduling parallel program tasks onto arbitrary target machines" [70] are examples of this approach.

As in the mapping problem, parallel algorithms are represented as task graphs. The size and complexity of the task graphs of real-life applications make this scheme impractical.

3.9. Future Work

We plan to use this method to develop domain-specific **problem solving environments** (PSE) and **application-oriented compilers** targeted to linear algebra and partial differential equations (PDE). PSEs and compilers based on this

method will be architecture-independent since the description of the processing environment is an independent parameter.

Although we focussed on templates based on parallel divide-and-conquer in this paper, the methodology presented here could be applied to other models as well. A case in point is the *sequential* divide-and-conquer (SDC), where dependencies exist between subproblems [71]. Templates based on SDC may be used to generate *pipelined* parallel programs. There are architecture-problem combinations where this approach will give the best results. Dynamic programming (DP), like the divide-and-conquer method, is a problem-solving strategy that solves problems by combining the solutions to subproblems [27]. But the structure of a DP-based template will be very different from that of a PDC-template. There are application domains that can benefit from the automatic parallelization that uses DP-templates as the seeds. We plan to extend the scope and solving power of our methodology by designing templates based on other paradigms, such as dynamic programming and sequential divide-and-conquer.

Most of the future work will be focused on three areas: performance prediction, the database of algorithm templates, and code generation. The performance prediction needs to be improved to take into account memory effects. Automatic generation of code for a specified target machine based on the template with best predicted performance is an area which requires further work.

The database of algorithm templates will be organized with a hierarchical structure. At the bottom, we will have basic linear algebra kernels, data distribution and communication primitives, and divide/combine functions. On top of this layer, non-trivial applications—such as *conjugate gradient method*, *two-dimensional FFT*, *banded-system solver*, and *eigensystem solver*—will be built. Numerical models and PDE solvers will form yet another layer. We believe this layered approach will have

the expressiveness to solve most problems in scientific computing, including irregular and unstructured problems.

4. SYSTEM INTEGRATION AND VALIDATION

An Architecture-Adaptable Problem Solving Environment for Scientific Computing

Santhosh Kumaran and Michael J. Quinn

Submitted to the *Journal of Parallel and Distributed Computing*

4.1. Abstract

The survival of parallel processing depends on a crucial factor: availability of programming systems which are effectively portable and easy to use. We extend our earlier work in architecture-adaptable parallel processing to develop such a parallel programming system. This paper discusses the design and implementation of the system. We have applied the system to solve problems in scientific computing on a diverse set of target processing environments. The applications include a *Kalman Filter*-based data assimilation scheme, a finite element model of regional ocean circulation, and stencil computing. The target platforms include multiprocessors, multicomputers, networks of workstations, and multiprocessor clusters. Results from these case studies show the expressibility, efficiency, and architecture-adaptability of the system.

4.2. Introduction

Is there a future for parallel processing? The improvement in single-CPU performance continues unabated while the independent manufacturers of parallel computers are becoming an endangered species [88]. Given the current unreliable nature of the parallel computing industry, it is only logical that most users are reluctant to use these systems.

To attract more users to parallel computing, it should be possible to develop application software independent of the target platform. Machine independence should be achieved without sacrificing efficiency. Additionally, tools should be made available to develop such software easily.

In [8], we introduced an architecture-adaptable parallel programming methodology. As a proof of concept, we employed our methodology to implement the conjugate gradient algorithm on a multiprocessor, a multicomputer, and a workstation network. This paper reports the design and implementation of a prototype code generator based on this methodology.

We have used case studies involving complex applications from the realm of scientific computing to validate our approach. The applications include a finite element model of regional ocean circulation [1], a Kalman Filter for data assimilation [12], and a stencil computation. The platforms we targeted include a symmetric multiprocessor (SMP), a multicomputer, a workstation network, and a network of SMPs.

The results show that the system has the expressive power to solve most problems in scientific computing. The architecture adaptability of the system is highlighted by the diverse range of our target platforms. The satisfactory level of

performance achieved across the target platforms shows that portability is achieved without sacrificing performance.

4.3. Methodology

Our methodology is built on the following three key ideas: (1) the use of a database of parameterized algorithmic templates to represent *computable* functions; (2) frame-based representation of processing environments; and (3) the use of an analytical performance prediction tool for automatic algorithm design.

The set of problems that can be solved efficiently using our approach is limited by the contents of the template database. The hierarchical structure of this database allows the solution of new problems by decomposing them into subproblems for which solutions already exist in the database.

Along with the problem to be solved, the user specifies the target architecture. The system maintains a database of parallel/distributed architectures. Two types of information are stored for each platform in the database: (1) information that determines the performance of a given template on the platform; (2) information required to generate code for inter-process communication.

The task of the system is to generate a parallel program which will run efficiently on the specified target platform to solve the specified problem. This task is accomplished in three steps:

1. **Selection of candidate templates.** A set of candidate templates is chosen from the template database to solve the specified problem.

2. **Searching for the best algorithm.** Each candidate template can have multiple realizations based on the values of the associated parameters. The space of "algorithms" to solve the problem is the union of the parameter spaces

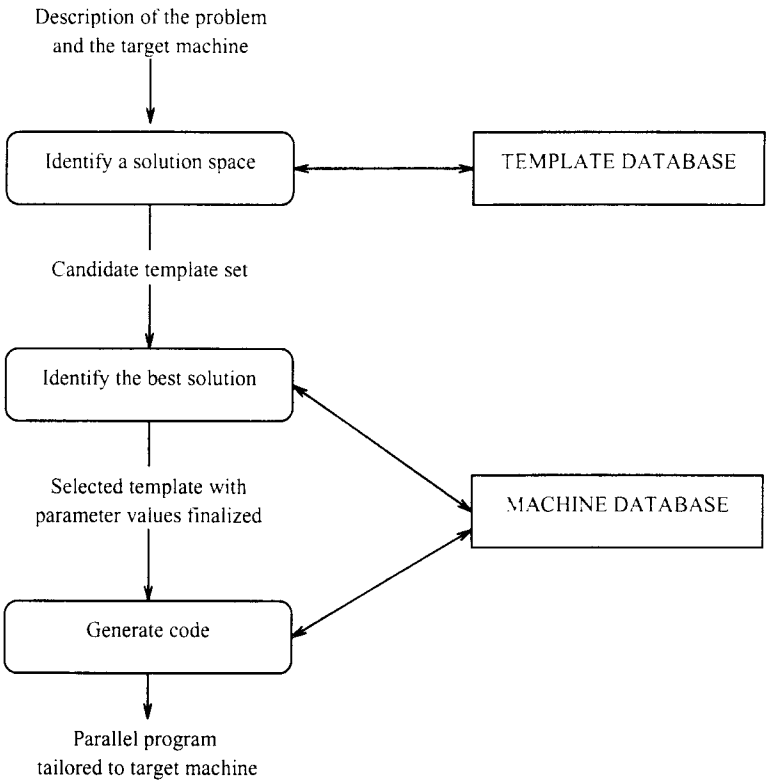

FIGURE 4.1. Schematic representation of our method for generating efficient parallel programs to solve a given problem on a specified architecture.

of the candidate templates. We use analytical performance prediction as an objective function to search this space for the best algorithm. At this point, the best algorithm is represented as a template with its parameter values specified.

3. **Code generation.** The last step is to generate code for the target platform to implement the best algorithm. A range of options is available for the format of the generated code. Our current implementation generates C source code, but the system can be modified to output code in any suitable intermediate representation, or even as a binary executable.

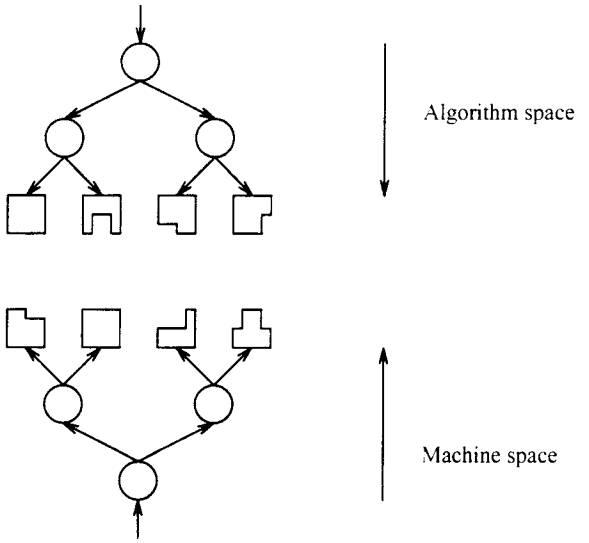

FIGURE 4.2. Schematic representations of the template and machine databases. While any target machine can execute any of the algorithms, choosing the right algorithm improves performance.

Figure 4.1 shows the method schematically. More details on the methodology can be found in [8].

4.4. System Components

In the previous section, we presented a procedural view of the system. Alternatively, the system can be viewed using an object-oriented paradigm. In this view, the template and machine databases are object-oriented databases representing the space of solutions and the space of target platforms respectively. Figure 4.2 shows these spaces schematically.

Given this representation, the system functionality is achieved by the following:

- Ability to generate abstract representations of the leaves of the algorithm tree from a top level node.
- Ability to search these leaves efficiently using a suitable objective function, given a specific leaf of the architecture tree.
- Ability to map the selected algorithm leaf onto the architecture leaf.

In this paradigm, the *template database* and the *machine database* become the most important components of the system. These databases are designed as class families. The system functionality—generating an efficient parallel program when presented with a problem instance and a target platform—is achieved by empowering the members of these families to generate a set of candidate algorithm leaves, to search this set to identify the best candidate, and to map the selected leaf to the target architecture leaf.

4.4.1. Template Family

Figure 4.3 shows the inheritance relationship for the template family. The template class has two subclasses, *base-template* and *composite-template*.

4.4.1.1. Base-Template

The base-template is the basic building block of an algorithm. A base-template *class* defines a generic methodology for solving a class of problems. A template *object* encapsulates a methodology for solving a specific problem. It has a well defined interface and an associated implementation. The interface specifies the problem being solved and the methodology used for solving the problem. The

implementation of the object deals with the details of the solution process. This is determined using the input parameters of the problem instance and the parameters of the target platform.

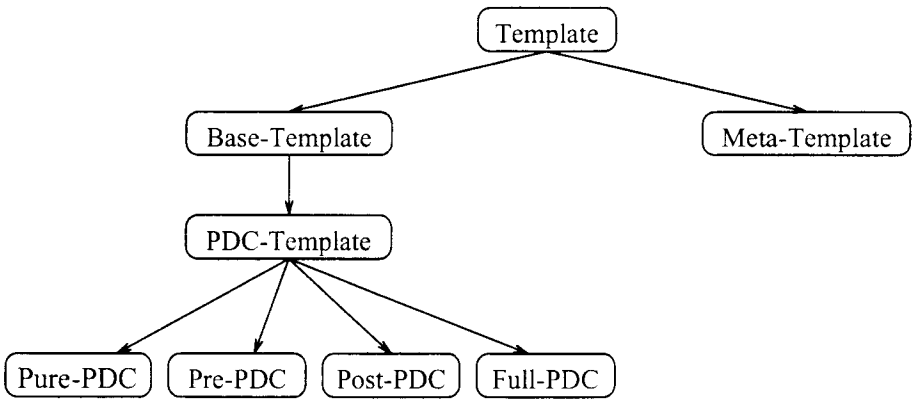

FIGURE 4.3. Inheritance relationships for template family.

In this paper, we will focus on base-templates designed using the parallel divide-and-conquer paradigm, named *PDC-Template*. These templates are based on the algebraic model developed by Mou and Hudak in [23].

PDC-Template: This class represents the parallel adaptation of the well-known divide-and-conquer problem solving paradigm [27]. A problem is solved in this paradigm by decomposing the original problem into a number of subproblems and solving the subproblems in parallel by the recursive application of the same procedure. Each instance of the class defines the application of this strategy to solve a specific problem.

Typically a divide-and-conquer algorithm has three phases:

1. A Divide phase in which the problem is decomposed into a set of smaller problems.

2. A Solution phase in which each of the subproblems is solved.

3. A Combine phase in which these solutions are combined to yield the result.

The above procedure is applied recursively to solve a problem where subproblems are smaller versions of the original problem. Infinite recursion is prevented using a base predicate, which is invoked at the beginning of the procedure. If the base predicate returns true, a base function is applied to solve the problem directly without any further division. It is easy to view this process as a tree with the root as the original problem and the base cases as the leaves.

Divide Phase and the Pre-adjust Functions: In the Divide phase, we divide a problem into a set of subproblems. For certain algorithms, it might be necessary to apply a function to the corresponding set of subdomains before the recursive application of the procedure to the individual subproblems. Such functions are named pre-adjust functions.

Combine Phase and the Post-adjust Functions: During the Combine phase, the solutions of the subproblems are merged to form a solution to the original problem. As in the Divide phase, we may have to apply a function to the subdomains; but in this case we do it after the procedure has been applied recursively to them. Such a function is called a post-adjust function.

Slots in the PDC template: To represent the above problem-solving methodology, the PDC template has slots to which we can attach the following:

- a divide function
- a combine function
- a pre-adjust function
- a post-adjust function
- a base predicate

- a base function.

Implementation of the PDC template: In addition to the slots given above, a template object has slots to hold parameter values. The implementation of a PDC template on a particular machine will be determined by the values of these parameters. The number of processors to use and the data decomposition scheme are the most important and universal parameters.

Although the specifics of the implementation of a template can change depending on the target platform and the problem instance, all implementations of the PDC template share a common strategy. This strategy combines the divide-and-conquer paradigm with the Single Program Multiple Data (SPMD) style of programming to provide an efficient implementation. The key to the efficiency is the new approach we introduced in mapping the PDC algorithm to the processing nodes. Details of this mapping are given in [27].

Subclasses of PDC template: There are four kinds of PDC-Templates: *Pure-PDC, Pre-PDC, Post-PDC,* and *Full-PDC.* These subclasses specialize the divide-and-conquer strategy further and break the PDC templates into four categories. The Pure-PDC has no adjust functions, representing templates with no communications at all. These are the *embarrassingly parallel* operations, such as the **saxpy** ($\alpha x + y$). The Full-PDC, on the other hand, has both pre-adjust and post-adjust functions, Cannon's matrix multiplication algorithm being an example [89]. Problems having only pre-adjust or post-adjust functions are categorized as Pre-PDC or Post-PDC accordingly. A simple example of a Pre-PDC operation is the hypercube broadcast algorithm, while several linear algebra operations can be represented using Post-PDC templates. Examples include dot product, row-oriented matrix multiplication, and matrix vector multiplication.

4.4.1.2. Composite-Template

A composite-template is composed of several templates. The parameters of the composite-template form a subset of the union of the parameters of its constituent templates. Most non-trivial applications are represented using composite-templates.

The conjugate gradient method [39] is an example of a composite-template with the following base templates as constituents:

- Column-oriented Matrix Vector Multiplication
- Block-oriented Matrix Vector Multiplication
- Row-oriented Matrix Vector Multiplication
- Dot Product
- SAXPY
- Vector Distribution
- Vector Concatenation

4.4.1.3. Member Functions of the Template Family

We briefly describe the two most important member functions of the template family:

1. **predictor:** The predictor function returns the performance prediction data in a machine-independent format. This is a polymorphic function, with the actual mechanism used for computing the prediction data depending on the type

of the template. A PDC-Template exploits the structure of the divide-and-conquer paradigm and relies on recursion to compute this data. A composite-Template simply invokes the *predictor* method of its constituent templates and compiles the results. The type of the target platform is passed as an argument to this method.

2. **codeGenerator:** This method of the template object is partially responsible for generating code to implement the algorithm that solves the problem efficiently. This algorithm is specified by the combination of three entities: the problem solving methodology encapsulated in the template, the set of function objects attached to the slots of the template, and the list of parameter values selected by the Search Engine. This method is invoked with the machine object as an argument. The machine object is responsible for generating machine-specific portions of the code, while machine-independent portions of code are generated by the template object. Our current implementation uses C as the base language.

4.4.2. Machine Family

A *machine class* represents an abstraction of a processing environment. The subclasses *HW* and *SW* encapsulate hardware and software features of the processing environment respectively. The *SM* subclass of HW represents shared-memory platforms, while the *DM* subclass represents the distributed-memory platforms. Within the distributed-memory machines, the *eXclusive-access* subclass defines machines with an exclusive-access communication network, such as Ethernet, while the *Nonexclusive-access* subclass represents architectures with an interconnection net-

work simultaneously accessed by several processors. Within the SW subclass, any inter-process communication library can be represented as a subclass.

Figure 4.4 shows a typical inheritance relationship for the machine family. We have defined a **cluster** as a processing environment with distributed memory, connected together using an exclusive access inter-connection network, and with PVM being used for inter-processor communication. Similarly, **SMP** is defined as a shared-memory system running the Solaris thread package. Any multicomputer, multiprocessor, or NOW can be represented using this framework. An obvious extension of this idea is to define hierarchical systems, where each level in the hierarchy is defined using the above framework.

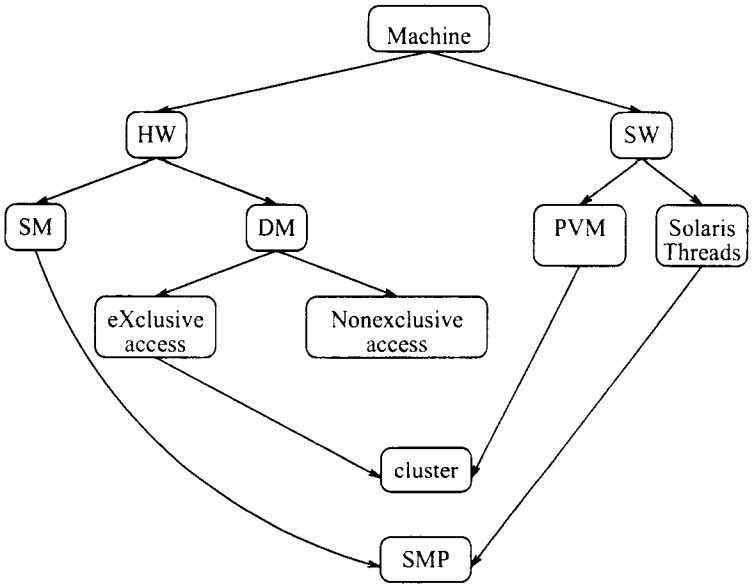

FIGURE 4.4. Inheritance relationships for a typical machine database.

Instances of the machine class represent specific parallel or distributed processing environments, such as a Meiko CS-2 or a cluster of IBM RS/6000 model 570 workstations connected together using an FDDI network.

Member functions: We briefly describe the two most important member functions of the machine family:

1. **timer:** The timer function computes the execution time of the template by combining the performance prediction data returned by the predictor function with machine-specific details.

2. **communicationHandler:** This polymorphic method outputs the machine-specific portions of the generated program. The behavior of the machine object in response to the invocation of this method depends on the type of the machine. A shared-memory machine will emit code using shared-memory primitives to handle the specified process interaction. A distributed-memory machine, on the other hand, will handle the communication request by generating message passing calls. The syntax of the generated message-passing calls or shared-memory primitives depends on the *SW* subclass to which the object belongs. If the machine is a cluster of workstations running under PVM, then PVM library calls are emitted. An object representing a shared-memory multiprocessor running Solaris will emit Solaris thread library calls.

   On shared-memory machines, communications will not result in explicit data movement. Instead, thread-specific pointers are simply set to point to different memory addresses. Data dependencies are satisfied by synchronizing the communicating threads using condition variables and mutual-exclusion locks.

4.5. System Integration

The two most important components of the system—the template family and the machine family—have already been presented. In this section, we briefly

describe the remaining parts of the system and show how these class families interact with the rest of the system.

4.5.1. Agent Classes

The rest of the system consists primarily of *agent* classes. Instances of these classes are the *agents of change*: they interact with the user, the databases, and with each other to realize the system functionality.

- **Template Selector:** This agent acts as the interface to the template database, with two major functionalities:

  1. Add new templates to the database.
  2. Retrieve all templates from the database with the capability to solve the problem specified by the user. (There could be several templates to solve the same problem. For example, there are three templates for matrix multiplication in our database.)

- **Machine Selector:** Like the Template Selector, the Machine Selector serves as the interface for the Machine Database. It facilitates addition of new machines to the database as well as retrieval of a machine object based on a machine identifier.

- **Search Engine:** This agent encapsulates the core of the system. It has a method, **solve**, which generates a program to solve a given problem efficiently on the specified target machine.

- **IO:** The IO agent is responsible for handling all input/output requirements of the problem. It generates code for reading application data into the template objects and outputting results of the computations.

Implementation of the solve method of the Search Engine: The Search Engine collaborates with the other agents and the template and machine objects to implement the solve method. Below we give the pseudocode for this method:

Method Name: Solve
Inputs:
Problem description,
Machine identifier,
Values of input parameters of the problem.
Outputs: A program that will solve the problem efficiently
on the target machine.
Pseudo-code:
Get a *machine* object from the **Machine Selector**.
Get a list of candidate *templates* from the **Template Selector**.
Invoke the *predictor* method of the **template** object
on each template to determine the best template and the
associated parameter values.
Invoke the *codeGenerator* method of the selected template
with the **machine** object as an argument to generate code.

Implementation of the performance prediction: Performance of a template on a target platform is computed in two steps. The process is set in motion when the Search Engine invokes the predictor method of the template object with the type of the processing environment as an argument. As mentioned before, the template object responds by filling up a container with the data for performance prediction in a machine-independent format. Once the performance prediction data are compiled, a *time* message is sent to the machine object to convert this data into normalized execution time. Thus, the performance prediction is done in two stages, thereby decoupling the template family from the machine family. This decoupling makes the two class families orthogonal, leading to several software engineering advantages. Figure 4.5 shows this two-step process schematically.

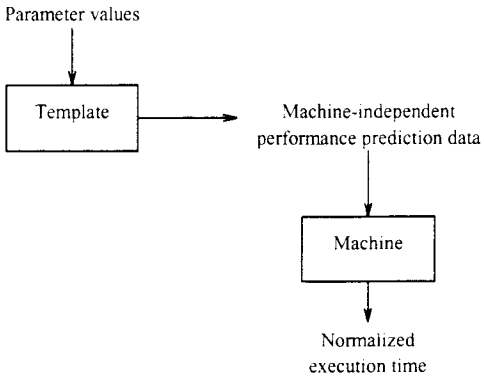

FIGURE 4.5. Schematic representation of our implementation of the performance prediction.

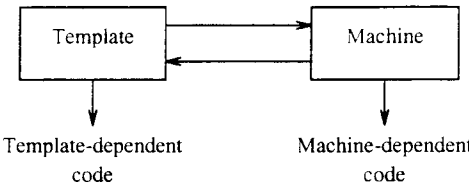

FIGURE 4.6. Schematic representation of our implementation of the code generation.

Implementation of the code generation: The emitted code is an SPMD program. The top-level structure of this program is well-defined and invariant with respect to the target platform as well as the specific problem being solved.

The Search Engine sends the *generate-code* message to the template object with the machine object as an argument to generate code. Note that the process interactions require machine-specific code. Since the templates are decoupled from the machines, this implies that a template object cannot generate code to handle process interactions. Instead, the template object simply sends a *handle-communication* message to the machine object when such code needs to be emitted. Figure 4.6 shows this process schematically.

4.5.2. The Participants and Their Roles

Our goal is to provide to the users a problem solving environment for high performance computing. This environment should be easy to use, powerful, and efficient. The power and efficiency of the environment will be discussed in later sections. Here we show how the users interact with the system and explore the *ease of use* aspect.

There are three types of participants in this project: system builders, system maintainers, and the system users. There is a very clear delineation of the roles of these participants. This three-tier view is pivotal in realizing the ease-of-use benefits.

The system builder is responsible for implementing the agent classes and the template and machine databases. The system builder provides tools for using and maintaining the system as well.

The system maintainer extends and updates the template and machine databases. It is his responsibility to add new machines and new templates to the database. For example, the system maintainer might extend an existing conjugate gradient template to develop a *preconditioned* conjugate gradient template and add it to the database. Since the parallel computing hardware and software are changing at a fast pace, the ability to extend the machine database is pivotal in achieving architecture adaptability. Consider a network of workstations (NoW) on Ethernet with PVM used as the message passing library. When this NoW is upgraded to use a single Myrinet switch [13] and Illinois Fast Messages (FM) [49] as the messaging layer, the system maintainer can easily generate a new machine class to represent this new computing environment. This is done by creating a **SW** subclass for FM

and inheriting from this new subclass as well as from the existing **HW** subclass **eXclusive-access**.

Finally, the user of the system, such as a computational scientist, simply invokes the *solve* method of the **Search Engine** with the problem description and the identifier of the target machine as arguments. The problem description will include a string that identifies the problem being solved and the values of input parameters. The exact nature of the user interface is beyond the scope of this paper, but we discuss the possibilities below and comment on their ease of use.

4.5.3. User Interface

The approach described above still requires some programming from the user. He has to write a small object-oriented driver program which should instantiate the agent classes and invoke the solve method of the Search Engine. Although this approach will give considerable flexibility and power, we feel it is important to provide a more user-friendly interface. There are at least two candidates for such an interface:

1. **MATLAB:** MATLAB is a well-known software package for numerical computation, data analysis, and graphics [4]. Complex numerical applications can be coded using the MATLAB notation. A front end could scan and parse the MATLAB programs, and generate an intermediate representation consisting of calls to high-level functions. For each function call, the *Search Engine* could search the database for matching templates.

   The main advantage of this approach is in providing a familiar interface to the computational scientist. But this will increase the complexity of the system considerably , since a combinatorial explosion of the parameter space is pos-

sible and the system must have the capability to deal with such situations. In most applications, branch-and-bound algorithms might be effective in pruning the search space and finding the best solution.

The MATLAB approach is somewhat at odds with the domain-specific methodology we espouse and the object-oriented paradigm we successfully used in designing and developing the system. The complexity of the programs written by the user will be determined by the MATLAB notation rather than the contents of the template database. Further, MATLAB uses a procedural notation and the software-engineering advantages of the object-oriented paradigm cannot be harnessed at the user level.

2. **Web-based interface (WEBLAB):** Alternatively, we could provide web-based tools to maintain and use the system. The template and machine databases can be distributed across the web. The user could invoke the system using a form-based interface from a web browser. The system would search the web for the template objects and machine objects, generate code, and run it on the target platform. This approach necessitates a template for each application, but it has the advantage of providing large scale software reusability.

4.6. Case Studies

In the earlier sections, we uncovered a system for problem solving on parallel and distributed processing environments. The design and development of this system was driven by our desire to make parallel processing "mainstream". The best way to gauge our success is, of course, to wait and see if our system indeed transforms the user base of parallel processing from the exclusive club of a few national laboratories to millions of mainstream users. Since that long a time-frame is

unacceptable for pragmatic reasons, we have tried to determine the effectiveness of our system through a few case studies.

In particular, we seek answers to the following questions:

- Is the system efficient?
- Does the system provide architecture adaptability?
- What is the expressive power of the model on which the system is built? In other words, what can it do and what is impossible?

We are using the case studies to answer these questions and thus demonstrate the worth of our approach.

Since the initial target application domain of our system is scientific computing, the foundation of our template database is a set of linear algebra kernels. We have built composite templates for complex scientific applications such as finite element model and Kalman Filter. Since stencil computations play an important role in scientific applications, we have explored the use of templates for them as well.

The target platforms include multicomputers, multiprocessors, and workstation networks. Recently, a new platform has emerged as a dominant high performance computing environment: *a network of multiprocessor workstations.* The applicability of our methodology in such an environment is studied in detail.

4.6.1. Linear Algebra Kernels

Below we list a sample of the linear algebra kernels in the template database:

- SAXPY

- Dot product
- Row-oriented matrix vector multiplication
- Column-oriented matrix vector multiplication
- Block-oriented matrix vector multiplication
- Row-oriented matrix multiplication
- Column-oriented matrix multiplication
- Block-oriented matrix multiplication
- Tridiagonal solver
- Banded-matrix vector multiplication
- Banded-matrix direct solver
- Iterative solver

The database has templates for the most frequently used data distribution primitives as well. These include:

- Vector distribution
- Vector concatenation
- Vector redistribution
- Matrix transpose

We will use matrix multiplication as an example and show how the templates are designed, how they get executed, and how they perform on various target platforms.

4.6.2. Matrix Multiplication Templates

Below we describe the matrix multiplication templates in the template database:

- **Row-oriented Matrix Multiplication** : $AB = C$. All data structures are distributed among the processors with row-contiguous distribution used for the matrices A and C and column-contiguous distribution used for the matrix B.

  Divide function: LR (Left-Right).

  Pre-adjust function: None.

  Base function: If in level 1, invoke sequential matrix multiplication; otherwise invoke a matrix multiplication template.

  Post-adjust function: (1) Swap values of B between partitions. (2) Recursively apply template to both partitions.

  Figure 4.7a shows the execution of the row-oriented matrix multiplication template on n processors. The unrolled execution sequence for four processors is shown in Figure 4.7b. Notice that the base function is another matrix multiplication template in level 1 and a sequential matrix multiplication routine in level 2. Figure 4.8 shows how the data structures at each processor change as execution proceeds.

- **Column-oriented Matrix Multiplication** : $AB = C$. The input data structures are distributed among the processors with column-contiguous distribution used for the matrix A and row-contiguous distribution used for the matrix B. On completion, matrix C is replicated on each processor.

  Divide function: LR.

Pre-adjust function: None.

Base function: If in level 1, invoke sequential matrix multiplication; otherwise invoke a matrix multiplication template.

Post-adjust function: Each processor gets the matrix C from its counterpart on the other partition and adds it to its own C.

- **Block-oriented Matrix Multiplication** : $AB = C$. The matrices are distributed block-wise among the processors arranged in a 2-D mesh.

  Divide function: LRTB (Left-Right-Top-Bottom).

  Pre-adjust function: If on RIGHT (partitions RT and RB), swap blocks of B with your vertical partner. If on BOTTOM (partitions LB and RB), swap blocks of A with your horizontal partner.

  Base function: If in level 1, invoke sequential Matrix Multiplication; otherwise invoke a matrix multiplication template.

  Post-adjust function: (1) Swap blocks of A with your horizontal partner. Swap blocks of B with your vertical partner. (2) Recursively apply Template to all partitions.

  Figure 4.9 shows how the data structures at each node change as execution proceeds.

Performance: In tables 4.1 and 4.2, we show the observed and predicted performance of the matrix multiplication routine on three diverse architectures—a multicomputer, a multiprocessor, and a LAN of workstations. The superlinear speedups on the multicomputer are due to cache effects. The source codes were generated automatically by the system in C with Elan Widgets, Illinois Fast Messages, and the Solaris thread package for the Meiko CS-2, workstation cluster, and the SPARCServer 20 respectively.

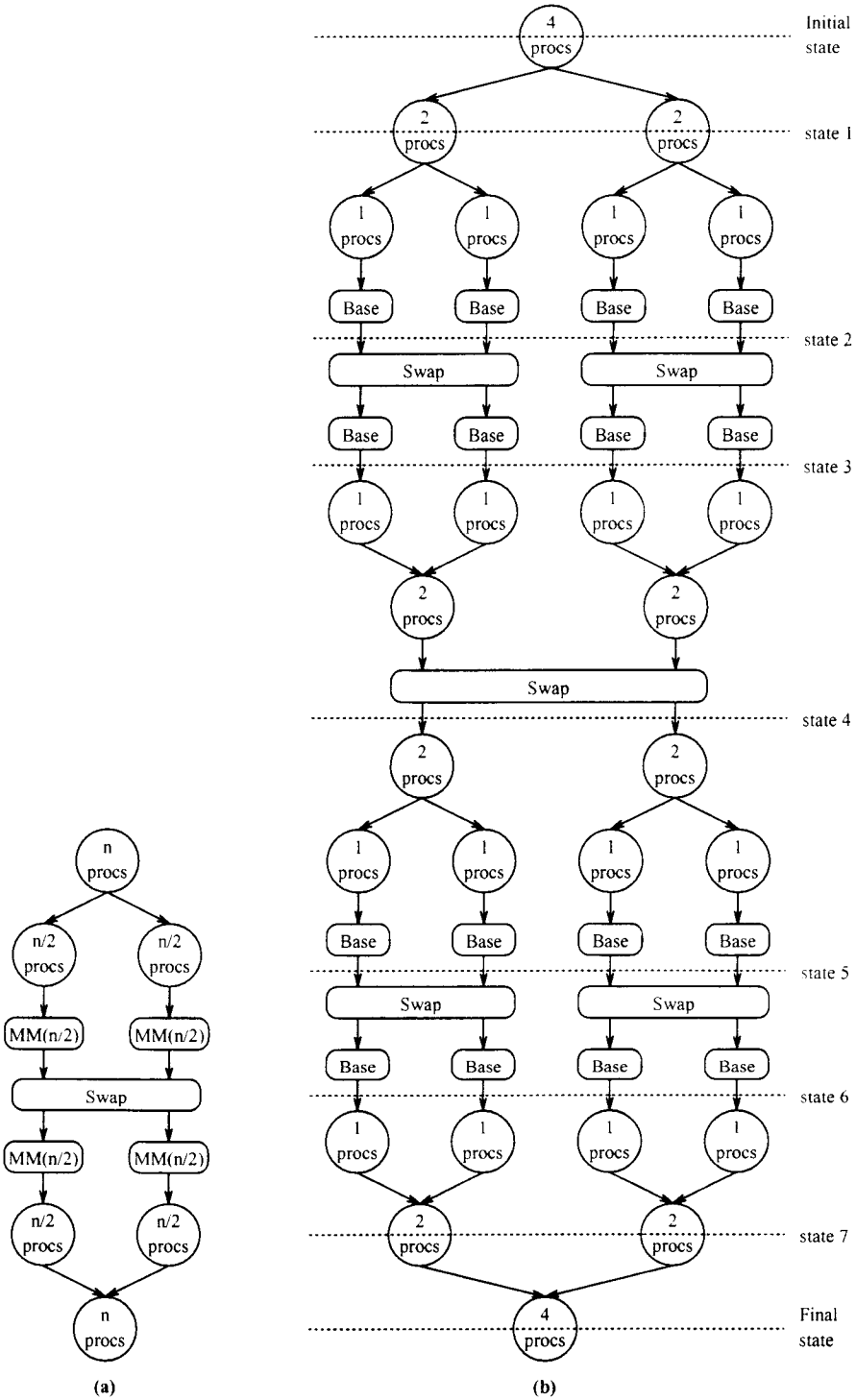

FIGURE 4.7. Schematic view of the execution of the matrix multiplication template. (a) Matrix multiplication on n processors. (b) Matrix multiplication on four processors after unrolling the recursion.

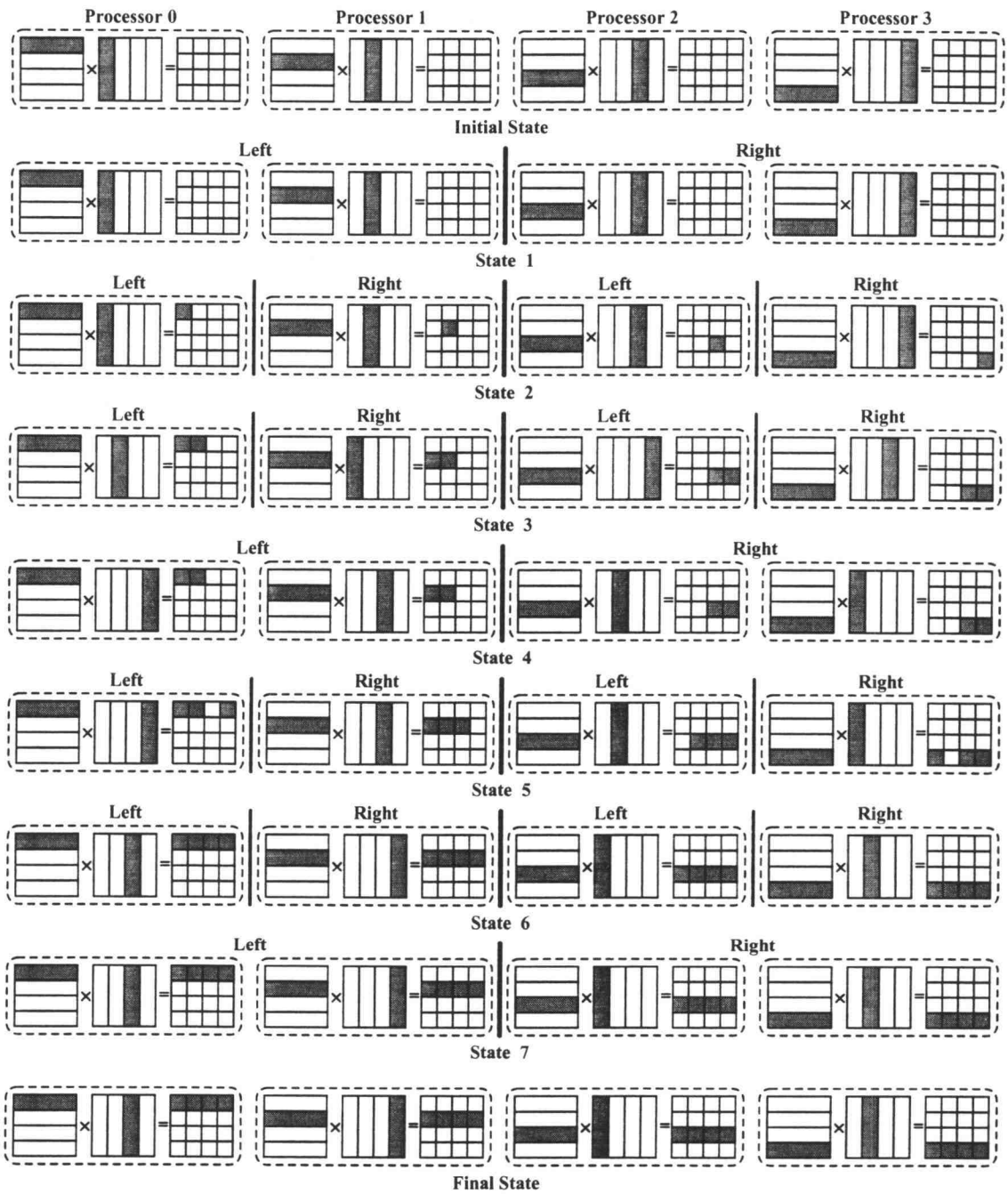

FIGURE 4.8. Snapshots of the data structures and processor partitions at the states labeled in the previous Figure. Shaded areas show the matrix blocks stored at a processor at the indicated state.

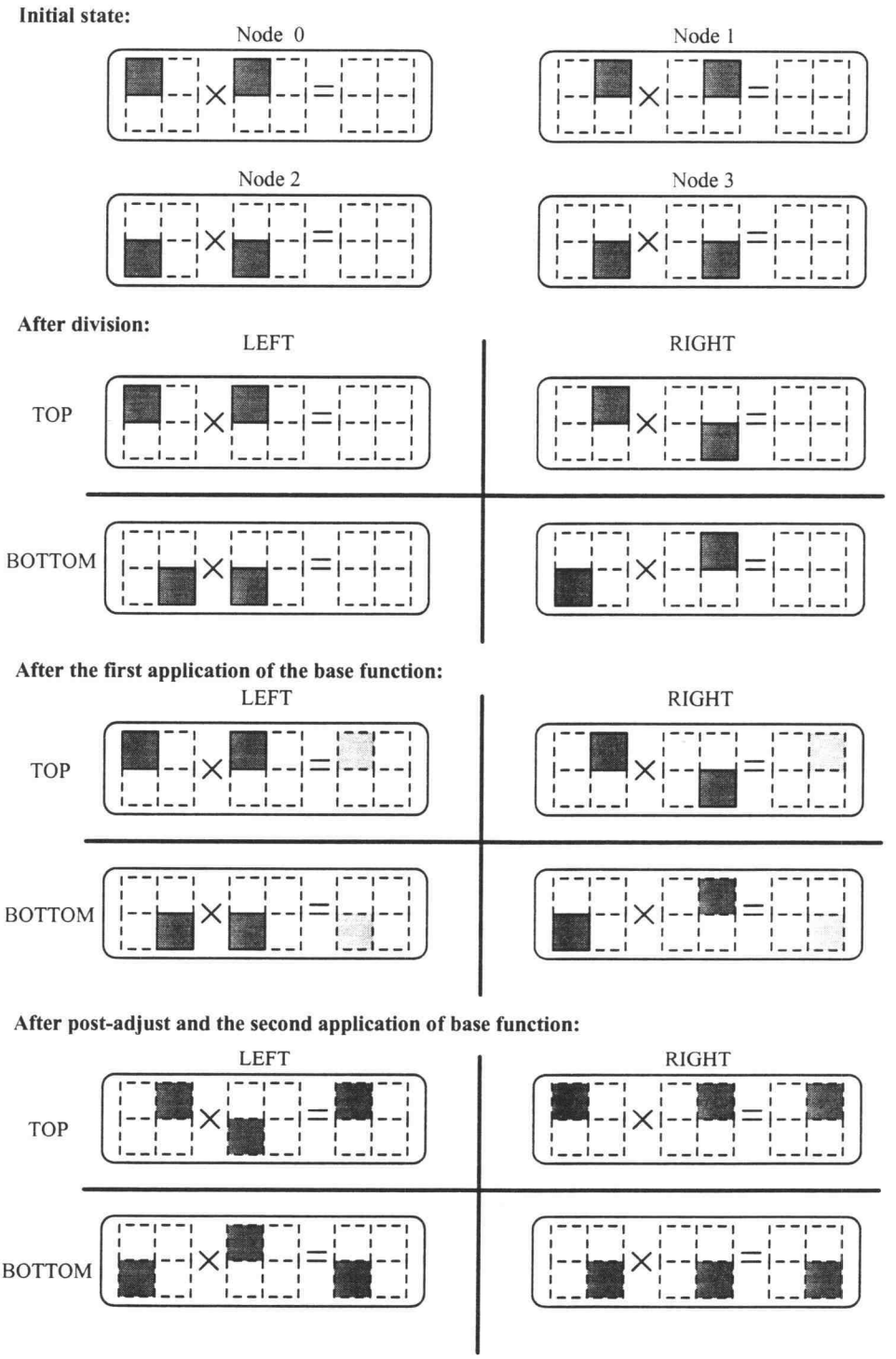

FIGURE 4.9. Snapshots of the data structures and processor partitions during the execution of the block-oriented matrix multiplication algorithm. Shaded areas show the matrix blocks stored at a processor at the indicated state. Lightly shaded blocks of the product matrix indicate partially computed results.

Machine	Processors				
	1	2	4	8	16
Meiko CS-2	1.0	2.1	4.4	8.1	14.2
CoW	1.0	1.9	3.8		
SPARCServer	1.0	2.0	4.0		

TABLE 4.1. Observed speedups of matrix multiplication on a Meiko CS-2, a cluster of IBM RS/6000 workstations on Myrinet, and a SUN SPARCServer 20. Matrices are of size 256×256.

Machine	Processors				
	1	2	4	8	16
Meiko CS-2	1.0	2.0	3.9	7.7	14.6
CoW	1.0	1.9	3.4		
SPARCServer	1.0	2.0	4.0		

TABLE 4.2. Predicted speedups of matrix multiplication on a Meiko CS-2, a cluster of IBM RS/6000 workstations on Myrinet, and a SUN SPARCServer 20. Matrices are of size 256×256.

4.6.3. Finite Element Model

As our next case study, we consider the parallelization of an ocean model [1]. The model uses quasigeostrophic equations to simulate regional ocean circulation. We use the finite element method to solve the resulting partial differential equations (PDEs).

4.6.3.1. Reduction into Subtasks

We begin by listing the important computational tasks in each step of the simulation process. These tasks and the PDC templates to represent them are given below:

- **Global matrix assembly:** The advection term in the model equations leads to a coefficient matrix which is a function of the state vector. The formation of the global coefficient matrix is an important task in the simulation, since this matrix needs to be recomputed at every time step. We present below a PDC template to represent the global matrix assembly:

  Divide function:
  Divide the processor pool into two equal partitions,
  a LEFT partition and a RIGHT partition.
  The FEM mesh is partitioned by contiguous strips
  among the processors.

  Pre-adjust function:
  Neighbor communication between the last processor on the LEFT
  and the first processor on the RIGHT to exchange data for
  the adjoining mesh points.

  Base function:
  Sequential global matrix assembly.

  Post-adjust function:
  None.

  Combine function:
  Combine the LEFT and RIGHT partitions.

- **Banded system solver:** The finite element discretization of the PDEs leads to three banded linear systems, each with 8432 unknowns and a bandwidth of 95. The solution of these systems is the most important task in the simulation, since the performance of the model is determined by the efficiency of the

solution algorithm. We use a domain decomposition method to solve the linear systems [90].

Domain Decomposition Method: Consider the problem of solving the linear system of equations $A\vec{x} = \vec{b}$, where A is the banded stiffness matrix, $\vec{x}$ is the unknown vector and $\vec{b}$ is the known right-hand side. To solve the system using the domain decomposition method, we partition the vector $\vec{x}$ (and $\vec{b}$) into three as $\vec{x} = \vec{x}_1\vec{x}_0\vec{x}_2$ (and $\vec{b} = \vec{b}_1\vec{b}_0\vec{b}_2$), where $\vec{x}_1$ and $\vec{x}_2$ are two contiguous chunks separated by $\vec{x}_0$. This algebraic partitioning of the vectors x and b corresponds to partitioning the gridded physical domain into two subgrids. The values of the gridded function x in the interiors of the subdomains are x_1 and x_2, and the components of x_0 are the values of x on the boundary which divides the full domain into subdomains. The size of the partition $\vec{x}_0$ is set to one plus the semibandwidth of the matrix. Now we rearrange the subvectors, forming a shuffled unknown vector $\vec{x}_0\vec{x}_1\vec{x}_2$ and a similarly juxtaposed right hand side $\vec{b}_0\vec{b}_1\vec{b}_2$. The modified system is given by the matrix equation below:

$$(4.1)\quad \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & 0 \\ A_{20} & 0 & A_{22} \end{pmatrix} \begin{pmatrix} \vec{x_0} \\ \vec{x_1} \\ \vec{x_2} \end{pmatrix} = \begin{pmatrix} \vec{b_0} \\ \vec{b_1} \\ \vec{b_2} \end{pmatrix}$$

The stiffness matrix transformation described above is graphically portrayed in figure 4.10. The structure of this new stiffness matrix is exploited to form the following solution scheme:

Step 1: Compute the "interface" variables $\vec{x_0}$ as:

$$(4.2)\quad \vec{x_0} = C^{-1}\vec{r_0}$$

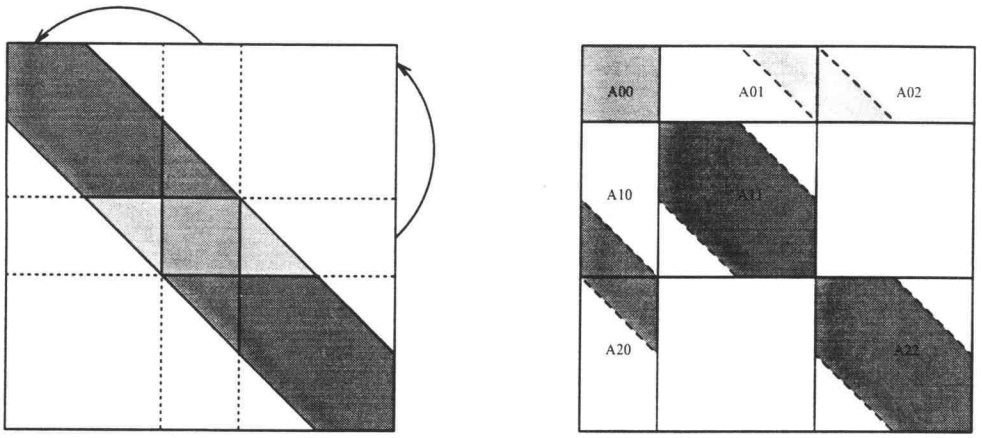

FIGURE 4.10. Matrix transformation for banded system solver. Original matrix is shown on the left and the transformed matrix on the right.

Step 2: Compute the "interior variables" $\vec{x_1}$ and $\vec{x_2}$ as:

$$(4.3)\quad \vec{x_i} = A_{ii}^{-1}(\vec{b_i} - A_{i0}\vec{x}_0) \quad \forall i = 1, 2$$

where:

$$(4.4)\quad \vec{r_0} = \vec{b}_0 - \textstyle\sum_{i=1}^{2}(A_{0i}A_{ii}^{-1}\vec{b_i})$$

$$(4.5)\quad C = A_{00} - \begin{pmatrix} A_{01} & A_{02} \end{pmatrix} \begin{pmatrix} A_{11} & 0 \\ 0 & A_{22} \end{pmatrix}^{-1} \begin{pmatrix} A_{10} \\ A_{20} \end{pmatrix}$$

An Optimization: Consider a banded system of the form:

$$(4.6)\quad \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & & & \\ a_{2,1} & a_{2,2} & \ddots & \ddots & & \\ \vdots & \ddots & \ddots & \ddots & \ddots & \\ & \ddots & \ddots & \ddots & \ddots & \vdots \\ & & \ddots & \ddots & \ddots & a_{n-1,n} \\ & & & \cdots & a_{n,n-1} & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_k \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

The solution vector x could be approximated by solving a smaller system of size $m < n$ as shown below:

$$(4.7)\quad \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & & & \\ a_{2,1} & a_{2,2} & \ddots & \ddots & & \\ \vdots & \ddots & \ddots & \ddots & \ddots & \\ & \ddots & \ddots & \ddots & \ddots & \vdots \\ & & \ddots & \ddots & \ddots & a_{m-1,m} \\ & & & \cdots & a_{m,m-1} & a_{m,m} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_k \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

and setting

$$(4.8)\quad \vec{x}_i = \begin{cases} \tilde{x}_i & i \leq m \\ 0 & i > m \end{cases}$$

Equations 3 and 5 in STEP 2 of the domain decomposition method can be approximated using this technique. This will give us a very good first guess

at the answer which can be further improved using iterative improvement. Although an accuracy analysis of this approximation is beyond the scope of this paper, we have successfully implemented this technique to analyze the meanderings of the Kuroshio, an intense current in North Pacific [1]. A single iteration was enough to restore the solution to the desired accuracy.

Template Representation of the Algorithm: We now present the algorithm using the divide-and-conquer notation.

Divide function:
Divide the processor pool into two equal partitions,
a LEFT partition and a RIGHT partition.
(In distributed-memory machines, this would automatically imply the division of data structures as well.)

Pre-adjust function:
None.

In the Conquer phase, we recursively solve the subproblems
$A_{11}\vec{x}_1 = \vec{b}_1$ on the left side and
$A_{22}\vec{x}_2 = \vec{b}_2$ on the right side.

Base function:
Sequential solver for banded linear system.
LU-factorization is done in the set-up phase.
Here we merely do the back substitution.

Post-adjust function:
STEP 1:
Left-side:
Compute $\vec{v}_1 = A_{01}\vec{x}_1$.
Right-side:
Compute $\vec{v}_2 = A_{02}\vec{x}_2$
STEP 2:
$\vec{r}_0 = \vec{b}_0 - (\vec{v}_1 + \vec{v}_2)$.
STEP 3:
Solve $C\vec{x}_0 = \vec{r}_0$
STEP 4:
Left-side:
Compute $\vec{d}_1 = A_{10}\vec{x}_0$
Solve $A_{11}\vec{f}_1 = \vec{d}_1$

$\vec{x}_1 = \vec{x}_1 - \vec{f}_1$

Right-side:

Compute $\vec{d}_2 = A_{20}\vec{x}_0$

Solve $A_{22}\vec{f}_2 = \vec{d}_2$

$\vec{x}_2 = \vec{x}_2 - \vec{f}_2$

Combine function:

$\vec{x} = \vec{x}_1\vec{x}_0\vec{x}_2$

Combine the LEFT and RIGHT partitions.

Since the coefficient matrix is a constant, so are the matrices C, A_{11}, and A_{22}. Since C is a full matrix of size w and $w \ll n$, where w is the semibandwidth and n the size of the system, it is possible to invert the matrix in the set-up phase and change the linear system solver in step 3 of the solver phase to a matrix-vector multiplication.

Using the approximation detailed in the previous section, we can drastically reduce the size of the systems to be solved in step 4. Further, this would enable us to replace the linear system solver by a matrix-vector multiplication, just as in step 3. Finally, since the non-zero component of $\vec{d}_x$ is only of length w, the multiplication involves a matrix of size $m \times w$ with a vector of size w, where m is the size of the reduced system.

- **Banded-matrix vector multiplication:** The right-hand sides of the linear systems are formed by a number of banded-matrix vector multiplications. We present below a PDC-template for this operation:

Divide function:

Divide the processor pool into two equal partitions,
a LEFT partition and a RIGHT partition.
The matrix is partitioned by contiguous rows
among the processors.
The vector is partitioned by contiguous chunks
among the processors.

Pre-adjust function:
Neighbor communication between the last processor on the LEFT and the first processor on the RIGHT to extend the subvectors.

Base function:
Sequential banded-matrix vector multiplication.

Post-adjust function:
None.

Combine function:
Combine the LEFT and RIGHT partitions.

Architecture Adaptability of the Algorithm: The template in the previous section is far from ready for implementation on a parallel computer. There are several details we need to include to make it a parallel algorithm suitable for solving a linear system. These details may be thought of as parameters of the divide-and-conquer template. To generate a parallel program from the template, we need to determine the appropriate values for these parameters. Ideally, we are looking for the values which will minimize the execution time of the resulting program on the target platform. In our methodology, architecture adaptability is achieved by choosing appropriate values for the parameters of the divide-and-conquer template based on the target platform. Essentially, we traverse a path from a generic method to a specific algorithm tailored to suit an architecture.

For example, the domain decomposition algorithm has three such parameters:

- Base predicate. This determines the depth of the divide-and-conquer tree. For the domain decomposition method, the number of subdomains is given by 2^d, where d is the depth of the tree. Best results may be obtained on coarse-grained machines by setting the number of processors the same as the number of subdomains. On the other hand, on a fine-grained, massively-

parallel architecture, each subdomain could be allocated to a pool of processing elements for optimum performance.

- Cut-off. This is the value of m in (4.7). As this value gets smaller, the accuracy of the solution suffers. To restore the desired accuracy, it may be necessary to do iterative improvement. For some architectures, a small cut-off value coupled with iterative improvement will produce an efficient implementation. For others, a large cut-off or no cut-off at all—thereby eliminating the need for iterative improvement— will be a better option.

- Granularity of the adjust function. A large number of options are available to us in implementing the post-adjust function defined in the divide-and-conquer template. As an example, consider step 1 of this function, which involves a matrix-vector multiplication on each partition. Notice we can perform the multiplication using any number of the processors in a partition. This gives us a new parameter k_{step1}, the number of processors we employ for computing step 1. Communication cost can be minimized if k is set to 1, which may be the right thing to do on a distributed environment such as the workstation network. Given a tightly-coupled parallel computer, on the other hand, it is advantageous to distribute the computation load across all the processors in a partition, as the communication overhead is comparatively small.

  Interestingly, each individual step in the adjust function can be implemented using a divide-and-conquer template [5]. Thus, k_{step1} is determined by the base predicate of this second-level template, nested within the first template.

Template for the Ocean Model: The set of computations involved in the ocean circulation simulation are captured in a parameterized composite-template, as shown in Figure 4.11.

Parameters:
K123: number of processors to be used for steps 1, 2, and 3
of the post-adjust function of the domain decomposition template.
K4: number of processors to be used for step 4
of the post-adjust function of the domain decomposition template.
M: Cut-off value to be used in step 4.

FEM-Template(K123,K4,M)
Invoke PDC-Template for global matrix assembly.
Do for each equation to be solved:
Invoke banded-matrix vector multiplication templates
to compute the right-hand side vector.
Invoke the banded-system solver template with
K123, K4, and M as parameter values
End **FEM-Template**.

FIGURE 4.11. A parameterized composite-template for a single iteration of ocean circulation simulation. Some details are omitted for clarity.

4.6.3.2. Performance

The ocean model was benchmarked on three different processing environments:

1. **Workstation Network:** A cluster of workstations (CoW) on Ethernet is our first processing environment. Each node in the cluster is an IBM RS/6000 workstation. We use PVM for message passing.

2. **Sun SPARCServer 20:** A multiprocessor with four processing nodes, this serves as the test platform for shared-memory machines. Multiprocessing is accomplished using Solaris threads.

3. **Meiko CS-2:** This is a multicomputer with vector nodes and a fast communication network with a fat-tree topology. The code was generated in C with the Elan Widget library used for communication.

Cluster size	Speedup	
	Predicted	Observed
1	1.0	1.0
2	1.8	1.7
4	3.1	2.9

TABLE 4.3. Predicted and observed speedups of ocean model on a cluster of IBM RS/6000 workstations on Ethernet.

Table 4.3 shows the performance of the model on the workstation cluster. The best performance is obtained when the computations in the post-adjust function are performed on a single node in each partition. The reduction of computation time from distributing these computations on several nodes is offset by the increase in the communication overhead in the cluster environment.

Table 4.4 shows the performance of the model on the SMP. The best performance is obtained when the computations in the post-adjust function are distributed among all processors in each partition. In shared-memory systems, there is no message-passing overhead for distributing work among the processors. Thus when more processors are added, the reduction in computation time more than offsets the increase in synchronization time. Hence the total execution time is reduced.

Table 4.5 shows the performance of the model on the Meiko CS-2. For best performance, computations in steps 1, 2, and 3 of the post-adjust function are performed on a single processor, while those in step 4 are distributed among all processors on each partition. On a multicomputer, the message-passing overhead is much smaller than on a cluster, and so the implementation benefits from distributing

Processors	Speedup	
	Predicted	Observed
1	1.0	1.0
2	1.9	1.8
4	3.2	3.4

TABLE 4.4. Predicted and observed speedups of ocean model on a Sun SPARC-Server 20.

Processors	Speedup	
	Predicted	Observed
1	1.0	1.0
2	1.9	1.8
4	3.4	3.2
8	5.5	5.3

TABLE 4.5. Predicted and observed speedups of ocean model on a Meiko CS-2.

the work in step 4 among the processors. The granularity of computations in steps 1, 2, and 3 is too small to benefit from parallel processing on this platform.

4.6.4. Hierarchical Systems

Since their inception, parallel processing architectures have continually evolved. Currently, there is a trend towards the merger of parallel processing and distributed computing. We believe the typical workstation of the future will be a multiprocessor. Local area networks, which are already omnipresent, will connect

several such multiprocessors together. The result will be distributed computing environments which are cheap, highly available and potentially fast [88].

Our approach to parallel processing has added significance for heterogeneous, hierarchical, and hybrid processing environments. For example, consider a processing environment consisting of multiprocessors connected by a local area network. Any efficient algorithm for solving a problem on this platform should take into account the vast difference in granularity within the system. This complicates further the already onerous task of parallel algorithm design. The implementation of the algorithm also becomes more complex since different mechanisms may have to be used for process interactions within the same program. Thus, a software system with the capability to automate the design and implementation of parallel programs has additional merit in such environments.

In this section, we describe the extension of our strategy to handle hierarchical systems. The natural hierarchical structure of the divide-and-conquer paradigm is exploited to adapt an algorithm to a multi-level, hybrid processing environment such as an SMP network. We discuss how template and machine objects are gracefully extended to automatically adapt an application to such an environment.

We present two case studies to illustrate this process: a stencil computation and a Kalman Filter implementation. Our target platform is a cluster of Sun SPARCServer 20s. We have experimented with two LAN technologies: Myrinet and Ethernet. Illinois Fast Messages (FM) are used for communication with Myrinet. With Ethernet, we use PVM for message passing.

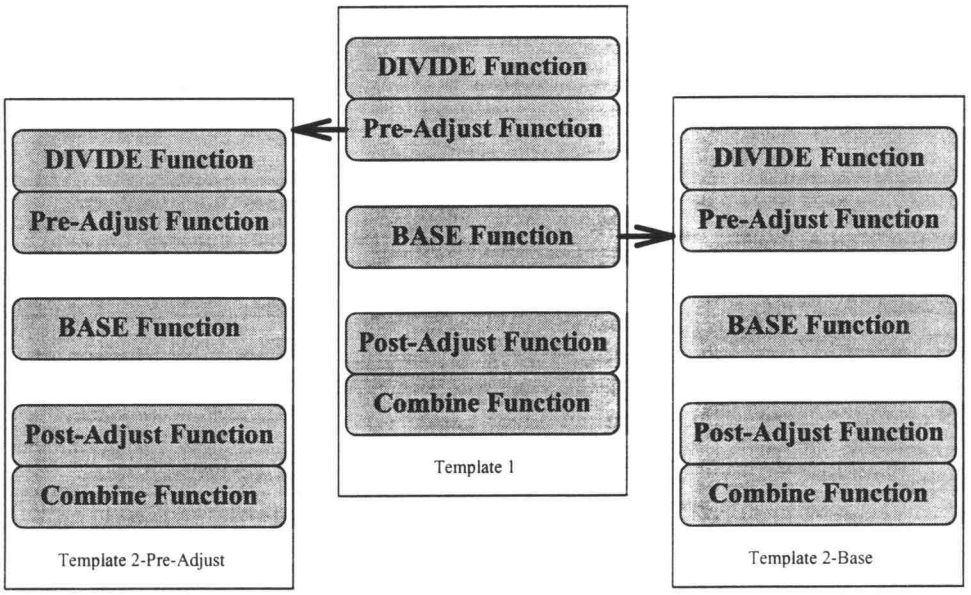

FIGURE 4.12. A hierarchical template.

4.6.4.1. Hierarchical Templates

The base function is strictly sequential in the PDC templates presented so far. If we remove this restriction and allow the base function to be another template, we get a hierarchical template as shown in Figure 4.12. For an SMP network, we use templates with two levels. The outer template corresponds to the network level, while the inner template corresponds to the multiprocessor.

4.6.4.2. Hierarchical Machine Objects

We define a hierarchical system using the class *Hierarchy*, as shown in Figure 4.13. The class Hierarchy has a finite number of levels; each level in the hierarchy is defined using a machine object. For example, a network of SMPs on Ethernet is

defined using a Hierarchy of two levels, with the top and bottom levels consisting of a cluster and an SMP as defined earlier.

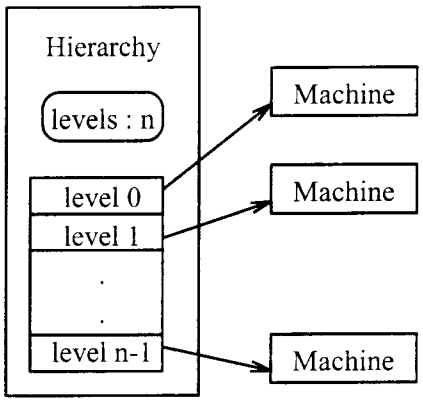

FIGURE 4.13. Representation of a hierarchical system.

4.6.4.3. Mapping of the Template to the Platform

At the outer level, a problem is mapped to a pool of network nodes and the subproblems generated from it are mapped to disjoint subsets of this pool, as shown in Figure 4.14. This process continues until there is only one node per partition. At this point, the second template is activated using the number of processors of the multiprocessor as the initial pool. Just as in the outer level, we continue the division of the processor pool and the subproblem until we end up with single processor partitions.

There are several important implications to this mapping:

1. When a problem is mapped to a set of network nodes, the data structures associated with it are distributed among the nodes' memories. We accomplish the division of the problem into subproblems by merely partitioning the set

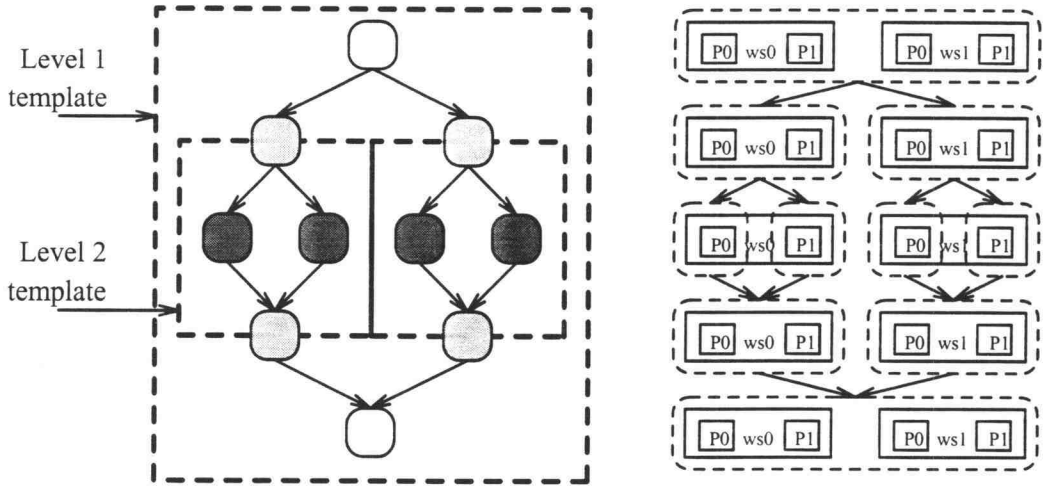

FIGURE 4.14. Mapping of the divide-and-conquer template to the SMP network. A two-level hierarchical template is shown on the left and the mapping on a cluster of two multiprocessor workstations, each with two processors, is shown on the right. The shaded squares denote the base cases.

of nodes to which it was mapped, without explicit movement of data among multiprocessor memories.

The divide function of the PDC template is responsible for the partitioning of the processing nodes. It might be necessary to modify the truncated data structures to preserve the integrity of the generated subproblems. We use the pre-adjust function to accomplish this. The solutions of the subproblems produce partial results, distributed among the processor memories. Final results are obtained by applying the post-adjust function to these partial results. The combine function is just the inverse of the divide function. It merges the partitions into a single set.

2. The shared memory of the multiprocessor is conceptually divided among its processors, and the algorithm execution proceeds just as in the outer level,

except that the processes communicate by reading from and writing into the shared memory.

3. This mapping leads to an SPMD implementation in which every node is active and doing useful work throughout the execution of the program. While the algorithm is executing at the outer level, all processors within a multiprocessor may or may not be active, depending on the nature of the adjust functions. The inter-node communications are carried out by a single thread of the multiprocessor, while computations can be done using all the available processors by spawning another template from within the adjust function, as shown in Figure 4.12.

4. Different templates can be used at the two levels. In general, the outer template is chosen to minimize the message passing overhead, while the best candidate for the inner template is the one which minimizes synchronization overhead.

All of these contribute directly to efficient problem solving on hierarchical platforms. It is important to note that *divide-and-conquer* is being used in this system merely as a methodology for designing the templates. Users do not write divide-and-conquer functions—they call higher-level functions like matrix-vector multiply or dot product. As shown above, the emitted code is not a divide-and-conquer program, but an SPMD program.

4.6.4.4. Performance Prediction Model

We presented an analytical model for predicting the performance of PDC templates in [8]. This model can be easily extended for hierarchical systems. We

present below the model equations for the specific case of an SMP network:

TERMINOLOGY:

$\vec{n}$	input size vector
$p1$	number of multiprocessors (level 1)
$p2$	number of processors in a multiprocessor (level 2)
k	number of subproblems generated by a divide operation
$\vec{n}_s$	input size vector of the subproblems
$f()$	sequential time
$S()$	speedup
T_{par}	parallel time
T_{comp}	computation time
T_{comm}	communication time
C_{comm}	communication time for combine phase
D_{comm}	communication time for divide phase
C_{comp}	computation time for combine phase
D_{comp}	computation time for divide phase
T_{par2}	parallel time in level 2
T_{comp2}	computation time in level 2
C_{comp2}	computation time for combine phase in level 2
D_{comp2}	computation time for divide phase in level 2
C_{sync}	synchronization time for combine phase (in level 2)
D_{sync}	synchronization time for divide phase (in level 2)

$$(4.9)\quad S(\vec{n}, p1, p2) = f(\vec{n})/T_{par}(\vec{n}, p1, p2)$$

$$(4.10)\quad T_{par}(\vec{n}, p1, p2) = T_{comp}(\vec{n}, p1, p2) + T_{comm}(\vec{n}, p1, p2)$$

$$(4.11)\quad T_{comm}(\vec{n}, p1, p2) = \begin{cases} 0 & (p1 = 1) \\ k \times T_{comm}(\vec{n}_s, p/k) + DC_{comm}(\vec{n}, p1, p2) & (p1 > 1) \end{cases}$$

$$(4.12)\quad T_{comp}(\vec{n}, p1, p2) = \begin{cases} T_{par2}(\vec{n}, p1, p2) & (p1 = 1) \\ T_{comp}(\vec{n}_s, p/k) + DC_{comp}(\vec{n}, p1, p2) & (p1 > 1) \end{cases}$$

$$(4.13)\quad T_{par2}(\vec{n}, p) = T_{comp2}(\vec{n}, p) + T_{sync}(\vec{n}, p)$$

$$(4.14)\quad T_{sync}(\vec{n}, p) = \begin{cases} 0 & (p = 1) \\ k \times T_{sync}(\vec{n}_s, p/k) + D_{sync}(\vec{n}, p) + C_{sync}(\vec{n}, p) & (p > 1) \end{cases}$$

$$(4.15)\quad T_{comp2}(\vec{n}, p) = \begin{cases} f(\vec{n}) & (p = 1) \\ T_{comp2}(\vec{n}_s, p/k) + D_{comp2}(\vec{n}, p) + C_{comp2}(\vec{n}, p) & (p > 1) \end{cases}$$

The equations above merely reflect the structure of the divide-and-conquer template and hence remain the same for all templates. Each template has additional expressions to compute the application-specific details.

We have incorporated the cache interference model developed by Lam *et al.* in [91] into our performance prediction tool to compute the cache misses. In addition to the cache size, the total number of cache misses are determined by a set of application-dependent parameters—called cache interference parameters—which are computed using the input size vector $\vec{n}$.

4.6.4.5. Case Study 1: Stencil Computations

Stencil computations play an important role in scientific applications. The data-parallel paradigm has been very successful in parallelizing stencil computations. How can we use the template approach for these computations? What do we gain from using templates in this case?

PDC Templates for stencil computations: A data-parallel language assumes a flat and homogeneous virtual processor grid. But, a hybrid processing environment, such as the SMP network, has a physical processor grid which is hierarchical with a vast change in granularity between levels. The cost of a neighbor communication will be determined by the weakest link in the grid, which is the shared network. As mentioned earlier, our PDC templates provide a natural and elegant mechanism to map the stencil computations onto a hierarchical grid. Partitioning of the domain among the processors is done in two levels: at the outer level, the subdomains overlap at the boundaries; at the inner level, partitioning is done without overlap. Figure 4.15 shows the partitioning schematically.

Below we present an example of an outer template and an inner template for a five-point stencil computation:

Outer Template:

Divide function:
    Divide the processor pool into two equal partitions,
        a LEFT partition and a RIGHT partition.
    The grid is partitioned by contiguous columns
        among the processors.

Pre-adjust function:
    Neighbor communication between the last processor on the LEFT
        and the first processor on the RIGHT to exchange data for
        the overlapping mesh points.

Base function:
    Invoke the inner template k times, where k is the

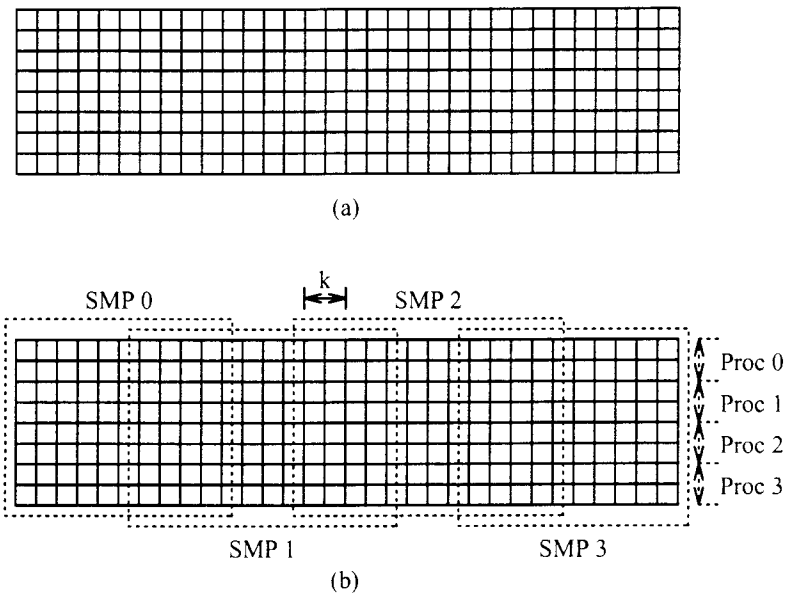

FIGURE 4.15. Multi-level splitting of domains in a hybrid processing environment. (a) Original grid. (b) Grid partitioned among the SMPs column-wise with overlapping and among the processor within an SMP row-wise without overlapping.

width of the overlapping region.
The subdomain shrinks by one grid spacing at both the left and right edges with each iteration.

Post-adjust function:
None.

Combine function:
Combine the LEFT and RIGHT partitions.

Inner Template:

Divide function:
Divide the processor pool into two equal partitions, a LEFT partition and a RIGHT partition.
The grid is partitioned by contiguous strips among the processors.

Pre-adjust function:
Neighbor communication between the last processor on the LEFT and the first processor on the RIGHT to exchange data for the adjoining grid points.

Base function:
Sequential stencil computation.

Post-adjust function:
None.

Combine function:
Combine the LEFT and RIGHT partitions.

The amount of overlap is an important parameter which should be tuned using the performance prediction data for the target environment. The decomposition of the grid (for example, row or block) is also a parameter for this application. Several templates might exist in the database corresponding to different decomposition schemes.

Notice that the communication in the pre-adjust function of the inner template will be replaced by a synchronization call when code is generated for a multiprocessor.

Performance: Tables 4.6 and 4.7 show the speedups of a 5-point stencil computation for varying subdomain overlaps on a 2-SMP cluster and a 4-SMP cluster respectively. We have used PVM for message passing between SMPs and Solaris threads for multiprocessing within an SMP. Each SMP has four processors. The grid size is 512×512. The results show that improvement in performance can be obtained by the very simple technique of overlapping the subdomains. The overlapping will increase the computation time, but the number of messages generated is reduced significantly. On systems where the message initiation overhead is significant, such as PVM, this results in the reduction of total execution time.

We should point out that our approach would facilitate more sophisticated solution techniques for grid problems on hybrid platforms. An example will be a multigrid approach, where an implicit direct method is used at the outer level and

overlap	Speedup	
	Predicted	Observed
1	5.8	5.3
2	6.1	5.6
4	6.3	5.8
8	6.3	5.7

TABLE 4.6. Predicted and observed speedups of a 5-point stencil computation on the SMP network for various values of grid overlap. There are two SMPs in the network, with four processors on each SMP. The grid size is 512×512. PVM is used for message passing.

an explicit iterative scheme is used at the inner level. This will limit the frequent neighbor communications required in a Jacobi-style iteration to the inner level and minimize the inter-SMP communication overhead. Additionally, different templates can be used to capture the data decomposition options and performance prediction can be used to select the best decomposition.

4.6.4.6. Case Study 2: Kalman Filter

The Kalman Filter is a widely used data assimilation scheme where the predictions of a numerical model are enhanced using field observations. In this scheme, the forecast error is estimated in a statistically optimal way by combining the predictions with observed data at each time step. We then correct the forecast field using the estimated forecast error.

overlap	Speedup	
	Predicted	Observed
1	7.6	7.7
2	8.3	7.8
4	8.8	8.2
8	8.9	8.8
16	8.7	7.9

TABLE 4.7. Predicted and observed speedups of a 5-point stencil computation on the SMP network for various values of grid overlap. There are four SMPs in the network, with four processors on each SMP. The grid size is 512×512. PVM is used for message passing.

The method: The Kalman Filter method is given by the following set of equations [12]:

TERMINOLOGY:

$\vec{u}_j^f$ forecast vector at time t_j

$\vec{u}_j^a$ analyzed vector at time t_j

$\vec{b}_j^o$ observation noise vector at time t_j

L_j transformation matrix at time t_j

P_j^f forecast error covariance matrix at time t_j

P_j^a analysis covariance matrix at time t_j

R_j observation error covariance matrix at time t_j

K_j gain matrix at time t_j

H_j matrix defining sampling scheme at time t_j

$$\vec{u}_{j+1}^f = L_j \vec{u}_j^a \tag{4.16}$$

$$P^f_{j+1} = LP^a_j L^T + Q_j \tag{4.17}$$

$$\vec{u}^o_{j+1} = H_j \vec{u}^t_{j+1} + \vec{b}^o_{j+1} \tag{4.18}$$

$$K_{j+1} = P^a_j H^T_{j+1} R^{-1}_{j+1} \tag{4.19}$$

$$\vec{u}^a_{j+1} = \vec{u}^f_{j+1} + K_{j+1}(\vec{u}^o_{j+1} - H\vec{u}^f_{j+1}) \tag{4.20}$$

$$P^a_{j+1} = (I - K_{j+1}H_{j+1})P^f_{j+1} \tag{4.21}$$

The explicit inversion of R, the observation error covariance matrix, can be avoided by an iterative scheme involving matrix vector multiplications [12]. Thus the entire scheme boils down to a series of dot products, matrix vector multiplications, saxpy operations, and matrix multiplications. Base templates exist in our database for each one of these computations. Thus a Meta-template can be used to represent the Kalman Filter in our system.

Since the computational cost of the Kalman Filter is dominated by matrix multiplication, the performance of the scheme depends heavily on the matrix multiplication code. The three PDC templates for matrix multiplication in our database give us nine possible implementations of matrix multiplication on a two-level hierarchical platform such as the SMP network. The outer template used for matrix multiplication also influences the data redistribution costs, since the same matrix appears at different positions in different equations.

Parameter space: A naive approach to template design can result in an unwieldy parameter space, but domain-specific knowledge can be used to prune this space significantly at the design stage. Below, we list the independent parameters of the Kalman Filter template:

1. P1MM: The number of nodes to use for matrix multiplication in the outer template.

2. P2MM: The number of processors to use for matrix multiplication in the inner template.

3. D1: Decomposition of the transformation matrix in the outer template. Possible values are R (row), C (column), and B (block).

4. D2: Decomposition of the transformation matrix in the inner template. Possible values are R (row), C (column), and B (block).

5. P2MVM: The number of processors to use for matrix vector multiplication in the inner template.

The number of processors to use in the outer template for matrix vector multiplication is not an independent parameter, but set to be the same as P1MM. As matrix vector multiplication is only an $O(n^2)$ operation, using two different values for these parameters will only increase the total execution time, since matrices will have to be redistributed between operations. This is a good example of pruning the parameter space at the template design stage.

So far, we have considered the parameter space of a template to be the product of the cardinalities of its independent parameters. But in complex composite-templates, it is possible to form the parameter space as the union of several subspaces, drastically reducing the size of the total space. Since the optimum number of processors to use for matrix vector multiplication in the inner template (P2MVM) can be computed independent of the value of P2MM, the parameter space of the inner template is composed of the union of two subspaces.

Composite-template: We present a stylized composite-template for Kalman Filter below. Matrix multiplications (MM), matrix vector multiplications (MVM), and data redistribution primitives are used as constituent templates. We

have not shown the invocations of the SAXPY operations in the composite-template for clarity.

Parameters:
P1, P2MM, D1, D2, P2MVM

Kalman-Filter-Template(P1, P2MM, D1, D2, P2MVM)
switch (D1)
case **Row**:
Invoke row-oriented MVM template
to compute the forecast vector.
Invoke row-oriented MM template as the first
step in computing the forecast error covariance matrix.
Transpose the product matrix.
Invoke row-oriented MM template to complete
the computation the forecast error covariance matrix.
Invoke row-oriented MM template
to compute the gain matrix.
Invoke row-oriented MVM template
to compute the analyzed vector.
Invoke row-oriented MM template as the first
step in computing the analysis covariance matrix.
Transpose the forecast error covariance matrix.
Invoke row-oriented MM template to complete
the computation of the analysis covariance matrix.
Transpose the analysis covariance matrix.

case **Column**:
Invoke column-oriented MVM template
to compute the forecast vector.
Invoke column-oriented MM template as the first
step in computing the forecast error covariance matrix.
Distribute the product among P1 nodes.
Invoke column-oriented MM template to complete the
computation of the forecast error covariance matrix.
Invoke column-oriented MM template
to compute the gain matrix.
Distribute the product among P1 nodes.
Invoke column-oriented MVM template
to compute the analyzed vector.
Invoke column-oriented MM template as the first step
in computing the analysis covariance matrix.
Distribute the product among P1 nodes.
Invoke column-oriented MM template to complete

the computation of the analysis covariance matrix.
Distribute the analysis covariance matrix.

case **Block**:
Invoke block-oriented MVM template
to compute the forecast vector.
Invoke block-oriented MM template as the first step
in computing the forecast error covariance matrix.
Transpose the product matrix.
Invoke block-oriented MM template to complete the
computation of the forecast error covariance matrix.
Invoke block-oriented MM template
to compute the gain matrix.
Invoke block-oriented MVM template
to compute the analyzed vector.
Invoke block-oriented MM template as the first step
in computing the analysis covariance matrix.
Invoke block-oriented MM template to complete
the computation of the analysis covariance matrix.

end switch (D1).

End **Kalman-Filter-Template**.

Performance: The target platform for this application is a cluster of SMPs connected together using a Myrinet switch. We use Illinois Fast Messages (FM) as the messaging layer. Table 4.8 shows the best values of the parameters for four different machine configurations and the corresponding performance data for a state vector size of 256.

4.6.5. Conclusions from Case Studies

The purpose of the case studies was to explore the expressiveness, efficiency, and architecture adaptability of our approach.

- **Expressiveness:** Case studies can only provide empirical evidence for the expressiveness of the approach. But the complexity of the applications we

Configuration	Processors	Parameters					Speedup	
		P1	P2MM	P2MVM	D1	D2	Predicted	Observed
2×1	2	2	1	1	C	R	1.9	1.9
2×2	4	2	2	1	C	R	3.7	3.8
2×4	8	2	4	1	C	R	6.9	6.2
4×4	16	4	4	1	B	R	9.9	10.4

TABLE 4.8. Predicted and observed speedups of ocean model on a cluster of Sun SPARCServer 20 SMPs on Myrinet. The configuration gives the SMP nodes in the network and the number of processors utilized on each SMP.

considered in this paper strengthens our earlier claim: our methodology has the expressive power to solve most problems in scientific computing.

A better way to address the expressiveness aspect is using computational complexity theory. We have shown in [6] that any parallel computation can be formulated using our computational model without changing its complexity class. In particular, if the original problem is in NC^k, then the formulation using our model is also in NC^k.

- **Efficiency:** The benchmarks show that the system delivers satisfactory performance for reasonable problem sizes. In fact, the biggest advantage of the system is its efficiency, which results from the following factors:
  - Better utilization of the memory hierarchy: There is a natural mapping between the hierarchical structure of divide-and-conquer and the memory hierarchy. By dividing problems until they fit into the L2 cache, memory

access times are considerably reduced. The same applies for out-of-core programs.

- Regular and structured communication patterns: The divide-and-conquer model of computation avoids expensive random communications. The PDC templates use only structured point-to-point communications.
- Base functions: The best sequential algorithm can be used as the base function to solve the subproblems.
- Automatic parameter optimization: By automating the search of the parameter space, the system can select the "best" implementation. A hand-coded program can perform better if the system search space did not include the implementation option represented by that program, the probability of which is very small for good template designs.

We compared the performance of the Kalman Filter code with a hand-coded, data-parallel, *non-divide-and-conquer* program. Both programs were run on the SMP cluster with PVM on Ethernet for communication. Not surprisingly, the template-based program outperformed the hand-coded version by a factor of three. This improved performance is primarily due to the ability of the system to search the parameter space and select the "best" implementation. A very similar performance improvement was shown by the template-based ocean model on the workstation network. A data-parallel conjugate gradient implementation exhibited practically no speedup at all, while the template-based approach gave a speedup of 2.9 on a cluster of four workstations. The latter splits the original problem into four subproblems and uses the best sequential algorithm to solve the subproblems, leading to better performance.

- **Architecture adaptability:** The most significant feature of our system is the ability to generate very different implementations on diverse architectures from a single template. The fact that we were able to handle gracefully a new and emerging processing environment, SMP network, is testimony to the architecture adaptability of the system.

4.7. Related Work

The philosophical foundations of our approach and those of the POET system developed at Sandia [92] have much in common. Both systems use a problem-oriented approach to parallel processing, as against the traditional algorithm-oriented approach, but the two systems differ significantly in the underlying methodology.

Karpovich *et al.* describes a parallel object-oriented framework for stencil algorithms in [93]. Their system is built using the Mentat Programming Language (MPL), an object-oriented parallel programming language [94]. Our focus is on developing a methodology to map problem instances to architectures automatically; we use object technology merely for implementing the methodology.

Chandy's group at Caltech [41] and Dongarra's group at Tennessee [42] are also investigating the use of "templates" for high performance scientific computing. Our method differs from these and other work on templates by introducing a novel approach to architecture adaptability, combining *parameterized templates* with analytical performance prediction.

Our performance prediction model is inspired by the work done by Clement and Quinn in predicting the performance of scalable data-parallel programs on multicomputers [36]. Parashar *et al.* have proposed the use of performance prediction

to improve the performance of parallel programs [40]. As part of a HPF/Fortran 90D application development environment, their performance prediction framework helps users in selecting appropriate compiler directives.

Several message passing libraries—most notably PVM [45], MPI [46], and p4 [47]—are in existence to facilitate parallel programming on network architectures. These libraries merely become accessories to our system, since our goal is the automation of algorithm design and implementation using the available programming tools.

4.8. Future Work

We believe that the work described here has immense potential to be the cornerstone of an enabling technology of great impact for scientific computing. Future research in this area will aim at delivering this technology to the user community and improving it based on user feedback. To this end, we plan to develop a **problem solving environment** (PSE) for scientific computing based on our approach. We plan to experiment with a MATLAB-style interface as well as a web-based interface for our PSE. Such a system will combine the ease of use of a PSE with the computing power of parallel and distributed platforms.

There are several ways we can extend the power of the system to increase its applicability:

- Heterogeneous processing environments. The structure of divide-and-conquer computations makes our system particularly well suited to heterogeneous processing environments. Our strategy will be to decompose the heterogeneous processing environment into a number of partitions, such that each partition is homogeneous. The original problem is divided into subproblems which are

mapped to the machine partitions. There are two advantages to this approach: first, selection of an algorithm to solve a subproblem can be made based on the machine partition it resides; second, expensive communication between machine boundaries can be minimized. The real challenge of *metacomputing* is getting performance without having to micromanage a large and complex processing environment. Our approach has the potential to meet this challenge.

- Irregular and unstructured problems. By assimilating existing mesh partitioning heuristics into the system, the scope of the system can be extended to include irregular and unstructured problems in computational science. For example, spectral bisection can be used to partition the unstructured mesh [95]. These partitions can be mapped to the LEFT and RIGHT machine partitions. This process can be repeated until single processor partitions result. At this point, the best sequential algorithm can be used to solve the problem.

- Adaptive computations. Several problems in computational science require dynamically varying algorithms, such as PDE solvers using adaptive finite element meshes. Recursive Hilbert space-filling curves can be used to impose a linear ordering on the finite elements in such meshes [96]. This ordering can be used to partition the mesh among processors. A formulation based on our PDC template is an obvious candidate to represent these computations.

4.9. Conclusions

I have combined object technology with a methodology for architecture-adaptable parallel processing to create a system for automatic generation of efficient parallel programs. The most important components of the system are two

object-oriented databases: a repository of parameterized algorithm templates and a framework for representing arbitrary target platforms. Below we list the highlights of the system:

- The templates need to be designed only once, since they are independent of the target platform.
- Code can be generated from the templates for a diverse set of hardware and software combinations.
- By generating source code in C with message-passing or thread-level library calls, existing compiler technology can perform code optimization and machine code generation.
- The template database can hold a diverse set of target platforms, and new architectures can be easily added to the database.

These are hard times for the parallel processing community. The major contribution of our work is to show that domain-specific problem solving environments can be designed to deliver machine-independent, efficient, and easy-to-use parallel processing. We conclude with these words of wisdom from Alan Perlis:

> "As always, we shall continue to avoid the Turing tar pit: being forced to use languages where everything is possible but nothing of interest is easy. [97]"

The research in models and systems for parallel programming has focused on Turing-equivalent languages. The power of such languages has proved to be a major obstacle in developing programming environments that appeal to a large user base. In this paper, we have shown that there is much to be gained (and little to lose) by using a

restricted computational model. Enough time has been wasted "stuck in the Turing tar pit". It is time to move on.

5. CONCLUSIONS

"There is only one basic way to deal with complexity:

Divide-and-Conquer."

–Bjarne Stroustrup

It appears we need only one basic mechanism to deal with parallel processing, at least for the restricted domain of scientific computing: Divide-and-Conquer.

5.1. Project Summary

We used the classic divide-and-conquer problem solving paradigm to design algorithmic templates. We introduced a new strategy to implement them on parallel and distributed platforms. By composing several base-templates, we formed composite-templates. Each template is parameterized to represent a rich implementation space. A hierarchical object-oriented database serves as a repository of these templates.

We devised a frame-based representation of the processing environment. An object-oriented database was designed to store these frames. Several examples demonstrate how this framework was used to represent diverse parallel and distributed processing environments.

We formulated the problem of architecture-adaptable parallel processing as model-driven mapping from the template space to the machine space. Figure 5.1 shows this mapping schematically. There are two stages in this mapping:

1. Select a suitable algorithm to solve the problem. This is done using analytical performance prediction.

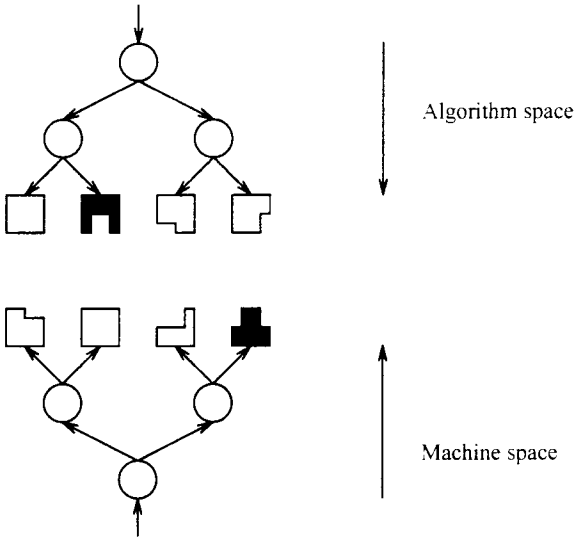

FIGURE 5.1. Architecture adaptability viewed as a mapping problem.

2. Generate code to implement this algorithm on the specified target platform.

We implemented the template and machine databases using object technology. We developed an object-oriented system to automate the mapping process. To validate the system, we used complex applications from the realm of scientific computing. The target platforms included multicomputers, multiprocessors, workstation networks, and SMP clusters. The results are very encouraging. The complexity of the applications tested is testimony to the power of the system. The range of our target architectures gives strong evidence of its architecture adaptability. The good performance of the programs generated by the system shows that portability is achieved without sacrificing efficiency.

5.2. Significance of this Research

Parallel processing is facing a software crisis. New and innovative ideas should be given due consideration, since the dominant paradigms have failed to realize the potential of parallel processing. In particular, there is a need for parallel processing environments which are architecture-adaptable, efficient, and easy to use.

This research departs from the mainstream of parallel processing research in several ways:

- Instead of the traditional algorithm-oriented approach to parallel processing, we use a problem-oriented approach.

- Most existing systems map from a Turing-computable set to a single machine. The system described in this dissertation maps from a restricted problem domain to a large machine space.

- We combine the divide-and-conquer problem solving strategy with the SPMD programming style, to realize efficient parallel implementations of divide-and-conquer algorithms.

- We use object technology only for the design and implementation of the system. The generated code is not in an object-oriented language. The current implementation uses C as the base language for code generation. The users do not write object-oriented programs to interact with our system. We anticipate providing either a MATLAB-style interface or a web-based, graphical interface.

The methodology developed in this research has the potential to make parallel programming easy, efficient, and architecture-adaptable. Commercial systems based

on this research could conceivably expand the user base of parallel processing by several orders of magnitude.

BIBLIOGRAPHY

[1] S. Kumaran, and R. N. Miller, "A Comparison of Parallelization Techniques for a Finite-Element Quasi-Geostrophic Model of Regional Ocean Circulation," *International Journal of Supercomputing Applications and High Performance Computing*, vol. 9, no. 4, pp. 256-279, Winter 1995.

[2] P. J. Hatcher and M. J. Quinn, *Data-Parallel Programming on MIMD Computers*, Cambridge, Mass.: The MIT Press, 1991.

[3] Gordon Bell, "Why there won't be apps: The problem with MPPs," *IEEE Parallel and Distributed Technology*, Vol. 2, Num. 3, 1994.

[4] Math Works, Inc., *The Student Edition of MATLAB: Version 4 User's Guide*, Prentice-Hall, 1995.

[5] S. Kumaran and M. J. Quinn, "Divide-and-conquer programming on MIMD computers," *Proceedings of the 9th International Parallel Processing Symposium*, 1995, pp 734-741.

[6] S. Kumaran and M. J. Quinn, "On Parallel Divide-and-Conquer," submitted to *IEEE Transactions on Parallel and Distributed Systems.*

[7] S. Kumaran and M. J. Quinn, "Architecture-Independent Parallel Programming using the Divide-and-Conquer Paradigm," in *Languages, Compilers and Run-Time Systems for Scalable Computers*, Chapter 6, pp. 71–84, Boleslaw K. Szymanski and Balaram Sinharoy (Editors), Kluwer Academic Publishers, Boston, MA, 1995.

[8] S. Kumaran and M. J. Quinn, "Towards architecture-adaptable parallel programming," *Scientific Programming*, to appear.

[9] S. Kumaran, R. N. Miller, and M. J. Quinn, "Architecture-adaptable finite element modeling: A case study using an ocean circulation simulation," *Proceedings of Supercomputing '95.*

[10] S. Kumaran, and M. J. Quinn, "Architecture-Adaptable Parallel Processing using Object Technology," *Parallel Object-Oriented Methods and Applications Conference*, Santa Fe, New Mexico, February, 1996.

[11] S. Kumaran, and M. J. Quinn, "Automatic Exploitation of Dual Level Parallelism on a Network of Multiprocessors," submitted to the *Fifth IEEE International Symposium on High Performance Distributed Computing*, Syracuse, NY, August 1996.

[12] R. N. Miller, "Theory and practice of data assimilation for oceanography," Reports in Meteorology and Oceanography, Number 26, Harvard University, Cambridge, MA., 1987.

[13] N. J. Boden, D. Cohen, R. F. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su, "Myrinet—a gigabit-per-second local-area network," *IEEE Micro*, 15(1), 1995, pp. 29-36.

[14] S. Pakin, M. Lauria, and A. Chien, "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet," *Proceedings of Supercomputing '95.*

[15] S. Kumaran and M. J. Quinn, "An Architecture-Adaptable Problem Solving Environment for Parallel and Distributed Computation," submitted to the *Journal of Parallel and Distributed Computing.*

[16] M. L. Minsky. 1967, *Computation: Finite and Infinite Machines*, Prentice Hall, Inglewood Cliffs, NJ, pp.160-162.

[17] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA., 1986.

[18] M. Wolfe. 1989. *Optimizing Supercompilers for Supercomputers.* The MIT Press.

[19] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, NY., 1979.

[20] R. Karp and V. Ramachandran, "Parallel Algorithms for Shared-Memory Machines," *Handbook of Theoretical Computer Science*, Volume A, J. Van Leeuwen, Ed., Elsevier Science Publishers, 1990, pp. 870-941.

[21] W. L. Ruzzo, "On Uniform Circuit Complexity," *Journal of Computer and System Sciences*, pp. 365-383, 1981.

[22] S. A. Cook, "A Taxonomy of Programs with Fast Parallel Algorithms," *Information and Control*, 64, pp. 2-22, 1985.

[23] Z. G. Mou and P. Hudak, "An algebraic model for divide-and-conquer algorithms and its parallelism," *The Journal of Supercomputing*, pp. 257-278, 1988.

[24] S. Bondeli, "Divide and Conquer: A new Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations," in *Proceedings of CONPAR 90 - VAPP IV*, Zurich, Switzerland, 1990.

[25] M. J. Quinn, *Parallel Computing: Theory and Practice.* McGraw-Hill Book Company, New York (1994).

[26] Thinking Machines Corporation, CMMD Reference manual, Cambridge, MA. 1993.

[27] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA., 1974.

[28] F. P. Preparata and J. Vuillemin, "The cube-connected cycles: A versatile network for parallel computation," *Communications of the ACM*, 8(5):300-309, May 1981.

[29] A. Gursoy, and L.V. Kale, "High level support for divide-and-conquer parallelism," *Proceedings Supercomputing '91.*

[30] Z. G. Mou and P. Hudak, "Parallel Programming in DIVACON," Tech. Report YALEU/DCS/TR675, Yale University, Dept. of Computer Science, 1990.

[31] A. J. Piper and R. W. Prager, "A High-Level, Object-Oriented Approach to Divide-and-Conquer," *The Fourth IEEE Symposium on Parallel and Distributed Processing*, December, 1992.

[32] L. Acker and R. Browning, "On Parallel Divide-and-Conquer," *the Seventh International Conference on Parallel and Distributed Computing Systems*, Las Vegas, Nevada, October, 1994.

[33] J. L. Bentley, "Multidimensional Divide-and-Conquer," *Communications of the ACM*, Vol. 23, pp. 214-229, 1980.

[34] E. Horowitz and A. Zorat, "Divide-and-Conquer for Parallel Processing," *IEEE Trans. oc Computers*, Vol. C-23, No. 6, June 1983, pp. 582-585.

[35] V. M. Lo, S. Rajopadhye, S. Gupta, D. Keldsen, M. A. Mohamed, B. Nitzberg, J. A. Telle, Xiaoxiong Zhong, "Mapping divide-and-conquer algorithms to parallel architectures," *Proceedings of the 1990 ICPP,* vol 3, pp. 128-135.

[36] M. J. Clement and M. J. Quinn, "Analytical performance prediction on multicomputers," *Proceedings of Supercomputing '93*, pp. 886-894, 1993.

[37] M. J. Quinn and P. J. Hatcher, "On the utility of communication-computation overlap in data-parallel programs," *Journal of Parallel and Distributed Computing*, To Appear.

[38] S. Hiranandany, K. Kennedy, and C.-W. Tseng, "Evaluating compiler optimizations for Fortran D," *Journal of Parallel and Distributed Computing*, 21(1), 1994, pp. 27-45.

[39] E. Polak, *Computational Methods in Optimization: A Unified Approach*, New York: Academic Press, 1971, pp. 44-66.

[40] M. Parashar, S. Hariri, T. Haupt, and G. C. Fox, "Interpreting the performance of HPF/Fortran 90D," *Proceedings of Supercomputing '94*, 1994, pp. 743-752.

[41] K. M. Chandy, "Concurrent program archetypes", *Proceedings of the 1994 Scalable Parallel Libraries Conference*, pp. 1-9, 1995.

[42] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, 1993.

[43] M. Metcalf and J. Reid, *Fortran 90 Explained*, Oxford Science Publications, Oxford, England, 1990.

[44] G. Alverson and D. Notkin, "Abstracting data-representation and partition-scheduling in parallel programs," *Proceedings of International Symposium on Shared Memory Multiprocessing*, 1991, pp. 138-151.

[45] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM 3 User's Guide and Reference Manual*, Oak Ridge National Laboratory, 1994.

[46] Message Passing Interface Forum, "The MPI message passing interface standard," Technical report, University of Tennessee, Knoxville, 1994.

[47] R. Butler and E. Lusk, "User's guide to the p4 parallel programming system," Technical report ANL-92/17, Argonne National Laboratory, 1992.

[48] L. A. Crowl, *Architectural Adaptability in Parallel Programming*, PhD Dissertation, University of Rochester, 1991.

[49] S. Pakin, M. Lauria, and A. Chien, "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet," *Proceedings of Supercomputing '95*.

[50] B. K. Szymanski, "EPL — Parallel Programming with Recurrent Equations", in B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, ACM Press, New York, 1991, pp. 51-104.

[51] M. Chen, Y. Choo, and J. Li, "Crystal: Theory and Pragmatics of Generating Efficient Parallel Code", in B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, ACM Press, New York, 1991, pp. 255-308.

[52] P. Hudak, and L. Smith, "Para-Functional Programming: A Paradigm for Programming Multiprocessor Systems," *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, ACM Press, New York, 1986, pp. 243-254.

[53] A. P. W. Bohm, D. C. Cann, J. T. Feo, and R. R. Oldehoeft, "SISAL 2.0 Reference Manual," Report CS-91-118, Computer Science Department, Colorado State University, Fort Collins, CO, 1991.

[54] K. Ekanadham, "A Perspective on Id", in B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, ACM Press, New York, 1991, pp. 197-254.

[55] P. Hudak, "Para-Functional Programming in Haskell", in B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, ACM Press, New York, 1991, pp. 159-196.

[56] A. S. Grimshaw, "Easy to use object-oriented parallel programming with Mentat," *IEEE Computer*, May 1993, pp. 39-51.

[57] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Yang, "Distributed pC++: Basic ideas for an object parallel language," *Object Oriented Numerics Conference*, 1993.

[58] Y. Yokote, M. Tokoro, "Experience and Evolution of Concurrent Smalltalk," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM Press, New York, NY, USA, pp. 406-415, 1987.

[59] A. A. Chien, "Efficient Concurrent Object Oriented Programming: Lessons from the Illinois Concert System," *Proceedings of the Parallel Object-Oriented Methods and Applications Conference*, 1994.

[60] L. V. Kale, "CHARM : Modularity and Reuse in Parallel Object-Oriented Software," *Proceedings of the Parallel Object-Oriented Methods and Applications Conference*, 1994.

[61] C. Kesselman, "Compositional C++: A parallel Object-Oriented Programming Language," *Proceedings of the Parallel Object-Oriented Methods and Applications Conference*, 1994.

[62] G. Agha, *Actors—A Model of Concurrent Computation for Distributed Systems*, MIT Press, 1986.

[63] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitct, K. Stanley, D. Walker, and R. C. Whaley, "ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance," University of Tennessee, Technical Report CS-95-283, March 1995.

[64] D. Quinlan and M. Lemke, "P++, a parallel C++ class library," *Object Oriented Numerics Conference*, 1993.

[65] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka, "Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools," *Object Oriented Numerics Conference*, 1994.

[66] R. Armstrong, "Use of frameworks for scientific computation in a parallel distributed environment," *Proceedings of the Parallel Object-Oriented Methods and Applications Conference*, 1994.

[67] J. F. Karpovich, M. Judd, W. T. Strayer, and A. S. Grimshaw, "A Parallel Object-Oriented Framework for Stencil Algorithms," *Proceedings of HPDC-2*, 1993, pp 34-41.

[68] J. Brown, "$MaTRiX+^{+}$: An Object-Oriented Environment for Parallel High-Performance Matrix Computations," *Proceedings of the Parallel Object-Oriented Methods and Applications Conference*, 1994.

[69] V. M. Lo, S. Rajopadhye, S. Gupta, D. Keldsen, M. A. Mohamed, B. Nitzberg, J. A. Telle, and Xiaoxiong Zhong, "Oregami: Tools for mapping parallel computations to parallel architectures," *International Journal of Parallel Programming,* vol.20, no.3, pp. 237-70, 1991.

[70] H. El-Rewini and T.G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *Journal of Parallel and Distributed Computing*,9(2), pp.138-153, 1990.

[71] Z. G. Mou, S. Anderson, and P. Hudak, "Parallelism in sequential divide-and-conquer," Technical Report YALEU/DCS/TR683, Yale University, Department of Computer Science, February, 1989.

[72] A. R. Clare and A. M. Day, "Experiments in the parallel computation of 3D convex hulls," *Computer Graphics Forum*, vol.13, no.1, pp. 21–36, 1994.

[73] W. R. Dyksen and C. R. Gritter, "Elliptic Expert: an expert system for elliptic partial differential equations," *Mathematics and Computers in Simulation*, vol.31, no.4–5, pp. 333–42, 1989.

[74] W. Luk, T. Wu, and I. Page, "Hardware-software codesign of multidimensional programs," *Proceedings IEEE Workshop on FPGAs for Custom Computing*, pp. 82-90, 1994.

[75] D. Gill and E. Tadmor, "An $O(N^2)$ method for computing the eigen system of $N \times N$ symmetric tridiagonal matrices by the divide-and-conquer approach," *SIAM Journal of Scientific and Statistical Computing,* 11(1), pp. 161:173, 1990.

[76] Yu Songnian, "An efficient parallel algorithm for the minimum spanning tree problem," *Chinese Journal of Computers,* vol.17, no.6, pp. 469–72, 1994

[77] A. M. Frieze, G. L. Miller, and Shang-Hua Teng, "Separator based parallel divide and conquer in computational geometry," *SPAA '92: 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 420-30, 1992.

[78] S. Hambrusch, Xin He, R. Miller, "Parallel algorithms for gray-scale image component labeling on a mesh-connected computer," *SPAA '92: 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 100-8, 1992.

[79] L. Cucu, M. Dragan, T. Jebelean, V. Negru, "Divide-and-conquer Voronoi diagram construction," *CG '94: 10th European Workshop on Computational Geometry,* pp. 5-7, 1994.

[80] C. Bischof, S. Huss-Lederman, Xiaobai Sun, A. Tsao, T. Turnbull, "Parallel performance of a symmetric eigensolver based on the invariant subspace decomposition approach," *Proceedings of the Scalable High-Performance Computing Conference*, pp. 32-9, 1994.

[81] R. van Liere, "Divide and conquer radiosity," *Photorealistic Rendering in Computer Graphics. Proceedings of the Second Eurographics Workshop on Rendering,* pp. 191-7, 1994.

[82] Wei Li, R. K. Kalia, S. De Leeuw, A. Nakano, D. Greenwell, P. Vashishta, "Parallel algorithms for molecular-dynamics simulations of Coulombic systems," *Materials Theory and Modelling Symposium*, pp. 267-72, 1993.

[83] P. Feautrier, "Some efficient solutions to the affine scheduling problem," *International Journal of Parallel Programming,* vol.21, no.6, pp. 389-420, 1994.

[84] A. Choudhary, R. Thakur, "Connected component labeling on coarse grain parallel computers: an experimental study," *Journal of Parallel and Distributed Computing,* vol.20, no.1, pp. 78-83, 1993.

[85] P. Cignoni, C. Montani, R. Perego, R. Scopigno, "Parallel 3D Delaunay triangulation," *Comput. Graph. Forum (UK),* vol.12, no.3, pp. C129-42, 1993.

[86] M. Wang, S. P. Vanka, "A parallel ADI algorithm for high-order finite-difference solution of the unsteady heat conduction equation, and its implementation on the CM-5," *Numerical Heat Transfer, Part B (Fundamentals),* vol.24, no.2, pp. 143-59, 1993.

[87] A. P. Willem Bohm, R. E. Hiromoto, "The dataflow time and space complexity of FFTs," *Journal of Parallel and Distributed Computing,* vol.18, no.3, pp. 301-13, 1993.

[88] Philip J. Hatcher, Robert D. Russel, Santhosh Kumaran, and Michael J. Quinn, "Implementing Data-Parallel Programs on Commodity Clusters," Spring School on Data Parallelism, Les Menuires (France), March 25-28, 1996.

[89] L. E. Cannon, *A cellular computer to implement the Kalman Filter Algorithm*, Ph. D. thesis, Montana State University, Bozman, MT, 1969.

[90] T. F. Chan, "Domain Decomposition Algorithms and Computational Fluid Dynamics," *CAM Report 88-25*, UCLA, Dept. of Mathematics, 1988.

[91] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," *Proceedings ASPLOS, 4*, 1991, pp. 63-74.

[92] R. Armstrong, "Use of frameworks for scientific computation in a parallel distributed environment," *Proceedings of POOMA '94*.

[93] J. F. Karpovich, M. Judd, W. T. Strayer, and A. S. Grimshaw, "A Parallel Object-Oriented Framework for Stencil Algorithms," *Proceedings of HPDC-2*, 1993, pp 34-41.

[94] A. S. Grimshaw, "Easy to use object-oriented parallel programming with Mentat," *IEEE Computer*, May 1993, pp. 39-51.

[95] T. F. Chan and W. K. Szeto, "On the Optimality of Median Cut Spectral Bisection Graph Partitioning Method," *CAM Report 93-14*, UCLA, Dept. of Mathematics, 1993.

[96] H. Sagan, *Space Filling Curves*, Springer-Verlag, 1994.

[97] A. J. Perlis, "Postscript," in "ACM Turing Award Lectures: The First Twenty Years, 1966 to 1985," ACM Press, 1987, p. 16.